Desarrollo de las bases de un sistema operativo inspirado en Plan 9

Trabajo de Fin de Máster

Curso 2019-2020

Mikel Cazorla Pérez

Dirigido por José Luis Risco Martín
y Juan Carlos Sáez Alcaide

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Calificación: 10

“Can you sum up plan 9 in layman’s terms?
It does everything Unix does only less reliably”

/usr/local/plan9/lib/fortunes

Desarrollo de las bases de
un sistema operativo inspirado en Plan 9

En este Trabajo de Fin de Máster se han sentado las bases de un sencillo sistema operativo de propósito didáctico para el PC (x86) de 32 bits denominado Mark 1. Para ello se ha creado un cargador de arranque a medida, una utilidad en línea de comandos llamada `f7disk`, una biblioteca mínima de C y un pequeño núcleo de 32 bits. También se ha diseñado un nuevo tipo de partición para instalar las imágenes del núcleo.

En el transcurso del desarrollo se han revisitado aspectos de la arquitectura elegida desde el punto de vista de la implementación, tales como la secuencia de arranque, el modo real y el protegido, la segmentación, la memoria virtual, el manejo de interrupciones y la planificación de hilos.

Se espera que Mark 1 fomente la experimentación construyendo nuevo software de sistema, y sirva como referencia y entorno de pruebas a otros estudiantes y entusiastas.

Palabras clave: *cargador de arranque*, *secuencia de carga*, *sistemas operativos*, *kernel*, *Plan 9*.

Development of the foundations of
an operating system inspired by Plan 9

This Master Thesis lays the foundation of a simple didactic operating system named Mark 1, targeting the 32-bit PC (x86). Four modules has been created to this effect: a bootloader, a command line utility named `f7disk`, a minimum C library and a small 32-bit kernel.

In the course of the project some aspects of the chosen architecture has been revisited from the implementation point of view, such as the bootstrap, real and protected mode, segmentation, virtual memory, interrupt handling, and thread scheduling.

Mark 1 is intended to encourage experimentation through the building of new system software, and serve as reference and test environment for other students and enthusiasts.

Keywords: *bootloader*, *bootstrap*, *kernel*, *operating systems*, *Plan 9*.

Índice de contenidos

Capítulo 1

Introducción

Los principales sistemas operativos de propósito general son complejos y están repletos de funcionalidades. Aunque el paso del tiempo ha pulido sus diseños, también ha resultado en una carga histórica que en ocasiones puede confundirse con una parte esencial o deliberada[1]. Por último, la continua incorporación de nuevas capas de abstracción hace que razonar sobre su funcionamiento interno sea cada vez más difícil.

En este Trabajo de Fin de Máster se dan los primeros pasos en el desarrollo de un sistema operativo libre y abierto de propósito educativo. Para ello, se pretende que el diseño sea lo más simple posible, se mantenga un conjunto de funcionalidades restringido y se reduzcan esas distracciones históricas en lo posible. También se espera que los resultados alienten a otros estudiantes para que experimenten diseñando y ensamblando nuevos sistemas operativos [1].

Este proyecto se inspira en otros que comparten ese espíritu didáctico. Igual que muchos de ellos se basan en UNIX, el que nos ocupa adopta algunos rasgos de **Plan 9**[2] [2], motivado por la observación de que su diseño es útil para ejemplificar las abstracciones de los sistemas operativos convencionales [3] y al ser una oportunidad para que los estudiantes que aún no lo conocen lo descubran.

[1] Sus sistemas de ficheros muestran claros ejemplos, tales como la mezcla heterogénea de sintaxis en MS-DOS o el uso habitual del directorio `/usr` («user») en sistemas herederos de UNIX.

[2] Originalmente conocido como *Plan 9 from Bell Labs*.

Cabe destacar que esta memoria no proporciona una explicación exhaustiva del diseño y funcionamiento de un sistema operativo. Para ello se hace referencia a otras fuentes [4]. La memoria describe los componentes actualmente desarrollados por el sistema operativo que nos ocupa, explicando los conceptos y técnicas más importantes que se han utilizado en el proceso. También recoge algunas de las decisiones de diseño tomadas y cómo se han afrontado los principales retos que han surgido en la realización este proyecto.

Objetivos del trabajo

El objetivo principal del proyecto es sentar las bases del nuevo sistema operativo para compatibles (x86), al que hemos llamado **Mark 1**. Todo el código debe ponerse a disposición de la comunidad para que se pueda usar como referencia, especialmente en el ámbito académico. Además debe ser fácil de modificar, compilar y ejecutar en un entorno básico que no exija una gran inversión.

Es importante que se pueda ejecutar Mark 1 sobre una máquina virtual, con el fin de que se puedan seguir fácilmente las instrucciones para la realización de pruebas y modificaciones. El software de virtualización elegido ha sido VMware Workstation Player 15, aunque se espera que el soporte se extienda próximamente a VirtualBox.

La primera versión de Mark 1 desarrollada en este proyecto cumple con los siguientes requisitos:

- Ofrecer soporte básico a la arquitectura x86.
- Cargarse en memoria desde un medio externo.
- Conmutar al modo protegido de 32 bits.
- Establecer un sistema mínimo de entrada y salida.
- Realizar una configuración básica de la memoria virtual.
- Ejecutar varios hilos en el kernel.
- Proporcionar una minibiblioteca de C que pueda usarse desde el núcleo.

Plan de trabajo

El proyecto ha consistido en el diseño e implementación de características o funcionalidades pertenecientes a cuatro grandes módulos: el cargador de arranque, la minibiblioteca de C, el núcleo y la utilidad `f7disk`. Estas se han ido incorporando en iteraciones semanales, subdivididas en etapas:

1. Estudio del material relativo a las características.
2. Diseño de un prototipo.
3. Implementación y experimentación.
4. Depuración del prototipo.

En el módulo del núcleo, algunas funcionalidades implementadas están fuertemente relacionadas entre sí (por ejemplo, tanto la configuración de la memoria virtual como el manejo de interrupciones avanzado dependen de la activación del modo protegido) de manera que su compleción progresiva desbloqueaba nuevas vías de desarrollo. Debido a eso, las iteraciones finalizaban revaluando la prioridad de las características pendientes y demostrando qué nuevas características eran practicables con base en el progreso realizado en la semana anterior.

Estructura de la memoria

- El capítulo 3 detalla las herramientas relevantes que han formado parte del **entorno de desarrollo**, enumera las razones por las que han sido elegidas y presenta la configuración empleada.
- El capítulo 4 describe la secuencia de carga centrándose en el diseño de dos componentes: el **cargador de arranque** y la **utilidad** `f7disk`.
- El capítulo 5 prosigue presentando la **estructura del kernel**, tanto a nivel binario como de código. Por primera vez, se combina código ensamblador con C, se introduce el concepto de controlador y se sientan las bases del resto de componentes.
- El capítulo 6 profundiza en la **minibiblioteca de C**. Su código está separado del núcleo para reutilizarse con facilidad.

- El capítulo 7 presenta el **manejo de interrupciones**, la **memoria virtual** y la **creación de hilos**.
- En los capítulos 8 y 9 se exponen las **conclusiones finales**, y se indican posibles líneas de **trabajo futuro**.

Capítulo 2

El entorno de desarrollo

La compilación del sistema está pensada para realizarse desde máquinas de tipo unix, habiéndose instalado una distribución de Linux[1] en la máquina anfitriona. El script de compilación está basado en Make, y puede hacer uso de algunas de las herramientas estándar[2]. En cualquier caso, las dependencias no son numerosas y sería posible reemplazarlas con facilidad si fuese necesario.

La mayor parte del código se ha escrito en C, el lenguaje de programación mayoritario para el desarrollo de sistemas operativos, y en el que están escritos aquellos que han servido de inspiración para el presente proyecto. Se ha considerado que, al ser un lenguaje conocido y bien probado en este campo, haría al código de Mark 1 más accesible. Para minimizar las dependencias y garantizar la portabilidad, C también se ha usado para el desarrollo de la utilidad `f7disk`, introducida en la sección 2.5.

Algunos fragmentos del código del núcleo de un sistema operativo no pueden escribirse usando lenguajes de alto nivel, ya que no pueden expresar ciertas instrucciones dependientes de la arquitectura. Es ahí donde interviene el uso directo de un lenguaje ensamblador, existiendo dos familias de ellos para x86, en función de su sintaxis: de *tipo Intel* (como en MASM) y de *tipo AT&T* (como en el de GNU).

Por último, es necesario algún mecanismo que permita probar el fun-

[1] Concretamente Ubuntu 18.04.3 LTS (Bionic Beaver).

[2] En el momento de la escritura, `cat` y `rm`. El conjunto exacto puede variar entre versiones, por lo que se recomienda consultar el fichero `/Makefile` directamente.

cionamiento de las imágenes del núcleo en el sistema anfitrión a medida que se está desarrollando. Existen dos opciones: los emuladores y las máquinas virtuales. Mientras que los emuladores son más simples, son ideales para pruebas rápidas y facilitan las labores de depuración, las máquinas virtuales suelen ser más cercanas al hardware real, y son esenciales para detectar errores que en un emulador podrían pasarse por alto.

A continuación se describen los componentes más importantes elegidos para formar parte del entorno de desarrollo y se exponen los motivos que han respaldado su inclusión.

2.1 Toolchain de C cruzado

El *toolchain* de C de GNU es ubicuo en el mundo Linux. Basado en **GCC** y **binutils**, las distribuciones suelen disponer de versiones precompiladas de fácil instalación gracias a algún sistema de paquetería (o incluso venir ya integradas). El script de Make puede valerse de una variable para usar el compilador predeterminado del anfitrión, sea o no GCC.

Aunque esta técnica se ha usado para para construir `f7disk`, se ha considerado que usar el *toolchain* el anfitrión es inapropiado para las imágenes del sistema operativo, porque las versiones precompiladas del de GNU están configuradas para generar binarios destinados a ejecutarse sobre sistemas idénticos al anfitrión. En teoría, esta configuración puede alterarse, pero el proceso es propenso a errores, «corrompiendo» las imágenes binarias generadas o rompiendo la compatibilidad del script.

Por ejemplo, el desarrollador es responsable de indicarle al compilador cuál es la arquitectura final. Este paso suele pasarse por alto o realizarse incorrectamente. Si la arquitectura del sistema anfitrión original fuese la misma que la de las imágenes, el error pasaría desapercibido durante mucho tiempo, hasta que alguien descargarse el código desde un anfitrión diferente para descubrir que el script de compilación contiene un error.

Por otro lado, el código de este *toolchain* es tremendamente complejo y evoluciona con rapidez [5]. Incluso si no se cometió ningún error en la configuración, un error en las versiones que acompañan al anfitrión

puede generar los mismos problemas anteriormente mencionados. Todo esto dificulta la reproducibilidad del proceso.

Es por eso que las imágenes del sistema se construyen con un *toolchain* cruzado construído ex profeso a partir del código fuente de GCC y binutils. Aunque Clang junto con LLVM era una alternativa interesante ya que es nativamente un *toolchain* cruzado, usarlos para evitar tener que compilar el *toolchain* no solucionaría todos los problemas mencionados. Además, solo serviría como aplazamiento, ya que sería necesario portar cualquiera de ellos a Mark 1 en caso de un futurible auto-hospedaje. GCC y binutils tienen más tiempo, son una solución muy asentada, y es más fácil encontrar información sobre como portarlos.

La principal ventaja de esta aproximación es que proporciona un mayor grado de aislamiento entre la imagen generada y el sistema anfitrión. Indirectamente, también reduce la complejidad de la configuración dado que, con el *toolchain* cruzado, no es necesario anular completamente la configuración de su arquitectura predeterminada: se puede construir a medida para la arquitectura deseada. Finalmente, el uso de una versión fija hace que el proceso sea reproducible.

El desarrollo de Mark 1 se ha realizado a partir de GCC 8.3.0 y binutils 2.32. Concretamente, se ha generado un *toolchain* para la arquitectura «i686-elf» y otro para la «x86_64-elf» con el objetivo de portar el sistema más adelante.

Estas son las opciones de la configuración adoptada [6]:

- `--target=i686-elf / x86_64-elf`
- `--disable-nls`
- `--with-sysroot (para binutils).`
- `--disable-werror (para binutils).`
- `--enable-languages=c,c++` (para GCC; solamente C es necesario).
- `--without-headers` (para GCC).

Además se ha utilizado la opción `--prefix="$HOME/local"`, ya que la variable de entorno `PATH` de la máquina de desarrollo estaba configurada para buscar binarios en el directorio `$HOME/local/bin`, manteniendo así

al *toolchain* cruzado completamente separado del resto de programas del anfitrión, y prescindiendo del uso de privilegios especiales.

El fichero Makefile de Mark 1 define algunas variables adicionales para configurar el *toolchain* cruzado. Estas variables están basadas en las que Make define implícitamente [7] para que resulten fácilmente comprensibles.

- `XCC (el compilador cruzado).`
- `XAR (el programa de archivado del` *`toolchain`* `cruzado).`
- `XCFLAGS =-ffreestanding -mno-red-zone $(CFLAGS)`
- `XLDFLAGS =-nostdlib`
- `XLDLIBS =-lgcc`

Solamente es necesario definir manualmente las dos primeras. Esto es posible exportando las variables en el proceso del intérprete antes de invocar a make:

```
$ export XCC=i686-elf-gcc XAR=i686-elf-ar
# ...
$ make
```

O bien desde la propia orden:

```
$ make XCC=i686-elf-gcc XAR=i686-elf-ar
```

Como puede apreciarse, el uso de un *toolchain* a medida no elimina la necesidad de especificar algunas opciones desde el fichero `Makefile`. Sin embargo, no se está anulando la configuración de su arquitectura predeterminada. Es por eso que el script solo es compatible con el *toolchain* cruzado y sería necesario modificarlo para que funcionase con otro.

Durante la compilación de la versión actual del núcleo (i686) solo es necesaria la primera (`-ffreestanding`), indicando que el programa no depende de la biblioteca estándar.

La segunda (`-mno-red-zone`) es relevante en la ABI del x86_64, donde las señales e interrupciones agregan un área de 128 bytes a la pila.

Por último, la opción del enlazador `-nostdlib` hará que los ficheros y bibliotecas de inicio no se utilicen. Aunque éste es el objetivo general, sí

conviene permitir al compilador usar libgcc, así que ahora debe agregarse de forma explícita. Además, ha de considerarse que éste puede intentar utilizar algunas funciones de las que ahora no dispone, y cuyas definiciones deben ser proporcionadas por el desarrollador. Esto se ha resuelto en la minibiblioteca de C (véase el capítulo 5).

2.2 Netwide Assembler (NASM)

En lugar de usar el mismo lenguaje ensamblador que el que genera el compilador de C, se ha optado por un lenguaje (y un ensamblador) separados: **NASM**. La elección de NASM sobre otras alternativas se debe a que cumple con todos los requisitos del proyecto, pero también viene acompañada de algunas ventajas adicionales.

En primer lugar, NASM puede generar código de 16, 32 y 64 bits «nativos», es decir, sin el uso de prefijos especiales en los códigos de operación. El soporte para 32 bits es necesario para todo núcleo que pretenda trabajar en ese modo, pero además, Mark 1 incluye bastantes fragmentos que trabajan en modo de 16 bits. En cualquier caso, disponer de los tres desde el mismo dialecto y ensamblador es conveniente a largo plazo.

En segundo lugar, los lenguajes ensambladores de tipo Intel son bien conocidos a nivel didáctico (Microsoft Macro Assembler o **MASM** es bastante popular en ese ámbito). Se ha considerado que el código resultaría más accesible usando uno de la misma familia.

En tercer lugar, la filosofía de su sintaxis la hace simple y transparente (a diferencia, entre otras, de la de MASM [8]). Por una parte, no intenta abstraer la programación artificialmente a través de una distinción firme entre el código y los datos. Por otra, el lenguaje no suele intentar adivinar la intención del programador cuando las instrucciones podrían resultar ambiguas, equilibrando la concisión con la explicitud. Se ha considerado que esto hace que el código sea más claro y legible.

Otros aspectos a tener en cuenta es que se trata de un ensamblador moderno y de desarrollo activo, que incluye un desensamblador propio

(`ndisasm`), y que su licencia libre y abierta [9] abre la puerta al autohospedaje.

En cuanto a su utilización, la mayoría de opciones para controlar el comportamiento del ensamblador van escritas en el código mediante directivas [10]:

```
[bits 32]
[org 0x00000000]
```

Gracias a esto, se ha generado código tanto de 16 como de 32 bits sin alterar el script de construcción.

Aparte de eso, se debe explicitar el formato del fichero resultante. Eso puede controlarse mediante la opción `-f` en la línea de órdenes. Mientras que el código del cargador de arranque está pensado para generar directamente un fichero binario, el núcleo se compone de muchos ficheros objeto en formato **ELF** (*Executable and Linkable Format*) que son enlazados posteriormente con la ayuda de un script:

```
$ nasm -f bin -o output.bin input.asm
$ nasm -f elf -o output.o input.asm
```

El el manual de NASM [11] puede encontrarse una explicación detallada de estas y otras opciones [3].

2.3 QEMU

Las pruebas durante el desarrollo se han realizado con el emulador de QEMU para las arquitecturas de 32 y 64 bits. Se ha decidido usar este emulador por familiaridad con su uso, pero se considera que BOCHS[12] es una alternativa prometedora.

Para simplificar la ejecución de QEMU y cargar una imagen de disco flexible, se recurrió al siguiente script[4]:

[3] Se aconseja activamente revisar la opción `-W+orphan-labels`.

[4] Si se hace clic en la ventana del emulador, éste capturará al cursor del ratón. La combinación para desbloquearlo (o bloquearlo de nuevo) es Ctrl + Alt.

```
#!/usr/bin/env bash
qemu-system-i386 -drive file=$1,index=0,if=floppy,format=raw
```

El argumento del script es la ruta a una imagen de disco flexible que el BIOS de QEMU cargará tal y como haría el hardware real. La especificidad sobre el tipo de la imagen (*floppy*, *raw*) no es caprichosa. Si no se indicase, el emulador restringiría parcialmente el acceso a ella como medida preventiva y el sistema no podría cargarse.

QEMU incluye un monitor[5] que permite inspeccionar el contenido de los registros y la memoria mediante una serie de órdenes [13]. La interfaz es tediosa y poco amigable, pero durante las primeras etapas no sobran las herramientas de depuración, por lo que conviene acostumbrarse a ella.

Los registros más importantes pueden examinarse a través de la orden `print`. Para los demás, puede recurrirse a la orden `info registers`. Finalmente, la memoria puede mostrarse con ayuda de la orden `xp /fmt`.

Esto puede acompañarse de algunos trucos, como detener al procesador mediante las instrucciones `cli` y `hlt`, o usando un bucle vacío infinito. Es posible comprobar si el programa se detuvo mostrando el contenido del puntero de instrucción, y aprovechar para examinar los registros y la memoria.

Desde el código escrito en C, conocer la convención de llamadas a las funciones es importante, porque puede explotarse para cargar un registro (en este caso `EAX`) con resultados numéricos y direcciones de memoria:

```
static void *getptr(void *ptr)
{
        return ptr;
}

...

getptr(addr);
for (;;);
```

[5] A diferencia de lo que indica la documentación de QEMU, para dirigirse al monitor se ha empleado la combinación Ctrl + Alt + 2 (o 1 para regresar al programa invitado).

El principio que hace que esto funcione se explica en la sección 4.1.

2.4 VMware Workstation Player 15

Al finalizar el desarrollo de cada característica de Mark 1 y comprobar su correcto funcionamiento sobre QEMU, dicha característica se probaba también sobre una máquina virtual Esto proporcionaba una oportunidad para hacer algunos ajustes y detectar errores que no se hubiesen manifestado en el emulador.

Inicialmente, se decidió usar VirtualBox[6], ya que es un proyecto de código abierto, puede usarse de manera gratuíta y suele estar disponible a través de los sistemas de paquetería usados por las distribuciones de Linux. No obstante, se descubrió que es la única plataforma de las utilizadas durante las pruebas cuyo BIOS no implementa la carga multipista, algo de lo que el cargador de arranque actual depende. Además, la máquina virtual no notifica que la carga ha fallado, y rellena el resto de la zona de carga con valores incomprensibles y erróneos.

Aunque las pruebas sobre VirtualBox fueron satisfactorias, las versiones actuales del sistema han crecido tanto que no pueden encajarse en las fronteras de una sola pista de la imagen de disco flexible. Este hecho obligó a cambiar a VMware hasta que el cargador de arranque pueda proporcionar la carga multipista por sí mismo.

2.5 La utilidad «f7disk»

Durante las primeras etapas del desarrollo del cargador de arranque, el script de Make se encargaba de generar una imagen de disco flexible clásica de «1.44 MB»[7] en la que instalarlo. Desde un sistema de tipo unix, crear la imagen se reduce utilizar la siguiente orden:

```
$ dd if=/dev/zero of=floppy.img bs=1024 count=1440
```

[6] Versión 5.2.34_Ubuntur133883 según `VBoxManage -version`.

[7] En realidad, esos discos flexibles tienen una capacidad de 1440 KiB. Según esta definición poco convencional, un «megabyte» equivaldría a 1000 x 1024 bytes.

Después, el script copiaba la imagen binaria del cargador al inicio de la imagen de disco. Aunque el procedimiento era sencillo desde el punto de vista del usuario, que podía hacerlo todo con una sola orden, también resultaba demasiado rígido y no estaba pensado para instalar varias imágenes binarias que tuviesen que convivir en el disco (en este caso, las del cargador y el núcleo).

Para resolver este problema y, al mismo tiempo, ceder al usuario el control sobre el medio de instalación, se diseñó un nuevo tipo de partición (descrita en la sección 3.3), y se desarrolló una nueva utilidad de línea de comandos: `f7disk`[8]. Ésta está pensada para utilizarse en combinación con una herramienta de particionado externa, a elección del usuario, y requiere cierta planificación para preparar a la unidad, tal y como se describe en la sección 3.3.2.

Una vez la partición ha sido inicializada, la gestión de la unidad corre a cargo de `f7disk`, que puede instalar nuevas versiones del cargador de arranque y las imágenes del núcleo siguiendo la configuración del usuario y respetando otras particiones presentes. Además, a través de los ficheros de dispositivo que proporcionan los sistemas de tipo unix, puede usarse sobre unidades físicas como discos duros o *pendrives*.

[8]El nombre hace referencia a `fdisk` y al descriptor de un byte que identifica a este nuevo tipo de partición: F7h.

Capítulo 3

El cargador de arranque

Al igual que con cualquier programa, para que la CPU pueda leer y ejecutar el código máquina del núcleo del sistema operativo debe primero poder encontrarlo en la memoria principal.

Como sabemos, los programas no residen en memoria mientras la máquina está apagada. La imagen binaria del sistema operativo está almacenada en algún soporte (habitualmente un disco duro), y es necesario que otro programa la *cargue* (es decir, la copie) en la memoria durante el arranque, dando así comienzo a la secuencia de arranque (o *bootstrap*). A este programa especial se le conoce como **cargador de arranque** (o ***bootloader***).

La paradoja de dicho proceso es que, para que sea posible llevarlo a cabo, el cargador de arranque debe también residir en memoria, de ahí que deba existir un primer programa que la arquitectura pueda cargar de manera autónoma. Pero eso también es problemático, ya que debe saber lo suficiente de él para poder cargarlo.

El arranque se complica aún más debido a las garantías de compatibilidad hacia atrás impuestas por la arquitectura. Conviene recordar que la evolución del PC/x86 se ha extendido por décadas, por lo que algunas limitaciones en su diseño se han mantenido con el fin de que programas muy antiguos sigan funcionando. Esto obliga a los sistemas operativos modernos a realizar una serie de configuraciones complejas durante el arranque, que además varían en función del fabricante.

Todos estos factores hacen que la secuencia de arranque sea altamente complicada.

3.1 Sobre el entorno de arranque

Existe un programa conocido como **BIOS** ('sistema básico de entrada y salida') almacenado en una ROM por el fabricante, en la placa base. Al comenzar la secuencia de arranque, el PC copia al BIOS en memoria automáticamente, se ejecuta el POST ('autocomprobación de encendido') y se realizan las configuraciones de arranque necesarias.

Por último, el BIOS selecciona cuál es la unidad de arranque y copia los primeros 512 bytes (conocidos como **«sector»**[1] **de arranque**) en la dirección de memoria 7C00h, justo antes de *saltar* allí. De esta manera, la arquitectura cede el control a un programa «externo».

Para que una unidad sea candidata para el arranque, el programa que contenga debe cuplir ciertos requisitos. El más importante es que debe terminar en el número mágico de dos bytes «AA55h»[2], conocido como **firma de arranque**. Al ser un patrón difícil de reproducir accidentalmente, puede utilizarse para distinguir si alguien ha colocado un programa en el primer sector, o si solo hay datos.

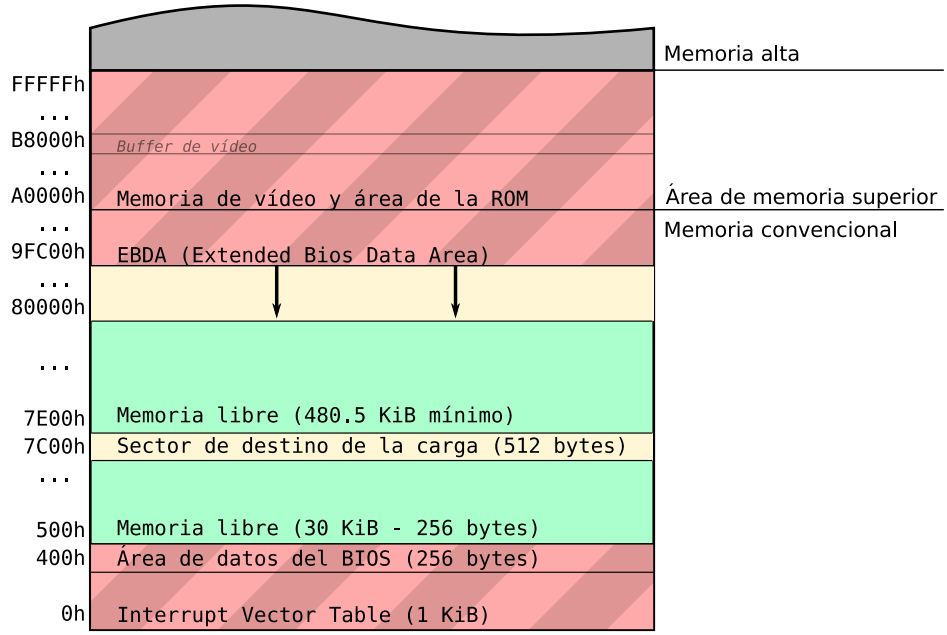
En el caso los discos duros, el sector de arranque contendrá al menos una tabla de particiones de 64 bytes justo antes de la firma. De este modo, dicho sector pasa a conocerse como **MBR** ('registro maestro de arranque'). Como se pudo descubrir al realizar pruebas sobre hardware real, el BIOS de algunos fabricantes también descarta aquellos discos que no contienen al menos una partición asignada en la tabla.

Debe notarse que el MBR contiene tanto códigos de operación como otros datos. Nuevas estructuras pueden ser situadas allá donde se considere oportuno, pero debe tenerse cuidado de que el procesador no intente ejecutar esos datos como si fuesen códigos de operación.

[1] El tamaño físico de los sectores de la unidad puede diferir.

[2] Teniendo en cuenta que el x86 es una arquitectura little-endian, en la representación en memoria primero va «55h» y después «AAh».

Figura 3.1: Mapa típico del primer MiB de memoria. Los 512 (200h) bytes del sector de arranque son copiados en la dirección ***7C00h****. Aunque la EBDA suele ocupar el último KiB de memoria convencional, el fabricante podría haber reservado espacio antes de esa zona (consulte la interrupción* `12h` *en el manual de su BIOS).*



Por ejemplo, sería un error situar una cadena de texto al comienzo del MBR, ya que esa parte será copiada en la dirección desde la que el procesador comenzará a ejecutar el «programa»:

```
[bits 16]

hello db 'Hello, World!', 0
jmp $

48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 00 <- 'Hello, World', 0
EB FE <- jmp $
```

Igualmente, hay datos tales como la firma de arranque que deben encontrarse en posiciones fijas. En lugar de contabilizar los bytes ocupados por los códigos de operación y rellenar los espacios vacíos a mano, es posible delegar esa tarea en el ensamblador:

```
...
times 510 - ($ - $$) nop
dw 0xAA55

... 90 90 90 <- ... nop nop nop
55 AA
```

Como puede apreciarse, la programación del sector de arranque implica razonar sobre su distribución interna, y a partir de cierto punto las herramientas no pueden ofrecer protección de lo que ese nivel de control supone. Aunque esta circunstancia es atenuada en el código del núcleo, contrasta con la mayoría de programas escritos en C.

Incluso tras haber cedido al procesador, el código máquina del BIOS permanece en la memoria principal (como puede apreciarse en la figura 3.1). El programa cargado tiene acceso directo a la memoria y puede sobreescribir las zonas ocupadas si así lo necesita. Claros ejemplos son el área de datos, el EBDA y la IVT. No obstante, otras zonas están mapeadas a componentes hardware y no se debe acceder a ellas a la ligera.

Por otro lado, el BIOS proporciona ciertas rutinas de servicio que pueden ser invocadas mediante interrupciones (instrucción `INT`). Algunas de ellas se ocupan de tareas comunes, mientras que otras desempeñan funciones que no pueden llevarse a cabo de otra manera. Es por eso que no conviene sobreescribir inmediatamente la memoria en la que el BIOS reside.

Inicialmente, el PC arranca en un entorno de compatibilidad conocido como **modo real**. Estas son algunas de sus características y restricciones:

- Permite ejecutar código escrito para el PC de 16 bits («como el 8086»).
- Las rutinas del BIOS solo pueden invocarse desde este modo.
- Carece de los mecanismos de protección hardware que la mayoría de sistemas operativos modernos demandan.
- Los modos de direccionamiento están limitados, imitando al 8086.
- Los registros extendidos son utilizables mediante un prefijo especial en el código de operación.

Existe una complicación adicional en el entorno de arranque: en muchas máquinas solamente puede accederse a los MiB impares de memoria. Esto sucede porque la línea de direccionamiento A20 (es decir, la que corresponde al vigesimoprimer bit de las direcciones de memoria) puede estar anulada por compatibilidad con los programas del 8086[3]. Este problema debe tratarse por separado y, desgraciadamente, no existe un método unificado que funcione en todas las máquinas.

3.2 El cargador de arranque de Mark 1

El núcleo del sistema preserva en memoria los códigos de operación del BIOS incluso tras concluir el arranque, con el objetivo de acceder a sus rutinas conmutando temporalmente al modo real. Eso motiva que delegue en el cargador el paso a modo protegido.

Para que el reparto de responsabilidades resulte evidente y mantener el control completo sobre ambos modos, se ha empleado un cargador de arranque a medida que respeta esas regiones de memoria, permite al núcleo evitar ciertas comprobaciones y copia su imagen binaria a partir de la dirección de destino 0D00h. A su vez, el núcleo respeta al cargador, y no reclama su memoria[4]. Al estar ambos bien integrados, se espera que en el futuro facilite el reinicio «suave»[5].

El cargador está pensado primeramente para ejecutarse desde un disco flexible o en modo de emulación. Esto significa que a menudo puede operar con otros soportes. Para ello, el cargador pide al BIOS información sobre la geometría de la unidad de arranque y recalcula las direcciones CHS a partir de las direcciones LBA. Aunque esta solución es algo restrictiva, se considera razonable en estas primeras etapas, al extender significativamente la utilidad del cargador más allá del entorno de pruebas.

[3]Debido a un antiguo bug introducido en el modo real del 80286.

[4]Se sugiere un diseño alternativo en el que el núcleo vuelva a copiar completamente al cargador al «retornar». Es importante que este sepa desde que unidad debe realizar la carga, así que algunos datos deben pasarse durante el arranque.

[5]Esta característica ya está implementada en el cargador. Véase la rutina de entrada `stage_2.softreset` en el fichero `/boot/stage2.asm`.

Las imágenes del sistema se almacenan en un nuevo tipo de partición especial. Aunque los discos flexibles no suelen utilizar particiones ni MBR, en este caso es una forma compatible y práctica de separar al sistema del resto de datos.

3.2.1 Estructura general y funcionalidad

Suponiendo que no se reserve espacio para más datos que la tabla de particiones y la firma de arranque, eso deja a los códigos de operación un espacio máximo de 446 bytes. Lo más importante que este programa ha de hacer es no desperdiciarlo. Introducir cadenas de texto tiene un altísimo coste en proporción a los códigos de operación, por lo que los errores se muestran al usuario mediante palabras clave y códigos.

Inspirado por los cargadores de otros sistemas, en cada etapa del arranque se imprimen letras en pantalla, formando la palabra «Boot». Esto es conveniente, porque si el programa se atasca o el procesador «se sale de su zona» debido a un error, es más fácil determinar dónde se produjo.

De todos modos, ese espacio no es suficiente. Al igual que en muchos otros cargadores, este está dividido en **dos etapas**. El programa del sector de arranque constituye la primera etapa, cuyas dos tareas más importantes consisten en garantizar que la línea de direccionamiento A20 está habilitada, y cargar a la segunda etapa. Esta se sitúa en los siguientes sectores de la unidad de arranque. Al conocer de antemano cuáles son y dónde se encuentran, la carga de esta etapa es más sencilla que la que requiere la imagen del núcleo, reduciendo el número de bytes invertido en los códigos de operación.

La segunda etapa tiene la oportunidad de realizar algunas comprobaciones adicionales, asegurando que el modo de vídeo ha sido configurado adecuadamente. También obtiene el identificador de la unidad de arranque guardado durante la primera etapa, y atisba en la tabla de particiones y la cabecera de la partición F7h (véase la sección 3.3) en busca de las imágenes del sistema.

La segunda etapa proporciona un mecanismo de reinicio «suave», y

por eso proporciona dos puntos de entrada. El primero y más sencillo conduce al código de reinicio, mientras que la primera etapa conoce la existencia de otro, destinado a un arranque completo.

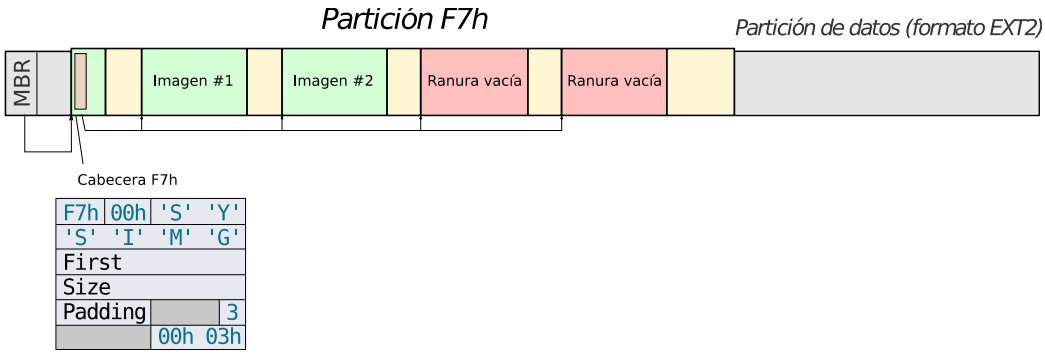
Puesto que el cargador no es desalojado de la memoria, el reinicio suave usa el identificador de la unidad previamente guardado, pero se asegura de volver a cargar el MBR para leer en la tabla, en previsión de que esta haya sido alterada durante el funcionamiento del sistema.

3.3 Particiones F7h

Las imágenes del sistema se almacenan en una partición de un tipo completamente nuevo a través de la utilidad `f7disk`. Las particiones de este tipo están subdivididas en de una a dieciséis **ranuras**, que no son sino espacios contiguos de memoria.

La relación entre las imágenes binarias y las ranuras es ilustrada por la figura 3.2. Por un lado, su uso evita que el cargador de arranque tenga que recomponer un sistema fragmentado, y por otro, dejan espacio a los usuarios para instalar sus propias versiones del sistema sin tener que sobreescribir la imagen binaria que están usando.

Figura 3.2: Diagrama de una partición F7h para imágenes del sistema con cuatro ranuras, dos de ellas ocupadas.



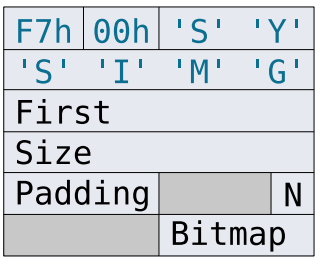
La ventaja de usar una partición especial para este cometido es que permite elegir la estructura de datos más conveniente para las imágenes

mientras coexisten con otros datos instalados en la misma unidad. Eso permite la compatibilidad con otras herramientas de particionado y formato a costa de consumir una partición principal. Al mismo tiempo, intenta amortizar dicho coste permitiendo instalar varias imágenes a la vez.

3.3.1 Estructura de las particiones F7h

Las particiones para imágenes del sistema se identifican por dos cosas: primero, el descriptor de un byte «F7h»[6], utilizado en la tabla de particiones, y segundo, una firma especial en la **cabecera**. Esta es una estructura de datos de veinticuatro bytes situada en el primer sector de la partición para almacenar datos sobre su configuración y estado.

Figura 3.3: Diagrama de la cabecera F7h correspondiente a la figura 3.2.



La firma ocupa los primeros ocho bytes de la cabecera, comenzando por el número mágico de un byte «F7h». Uno de los objetivos de incluirla es realizar una comprobación desde la segunda etapa del cargador de Mark 1 para garantizar que la partición se encuentra realmente instalada en la unidad de almacenamiento. Sin embargo, ante la creciente escasez de descriptores libres, el principal motivo de su existencia es no acaparar al que se ha elegido para poder reutilizarlo luego. Es por eso que la firma contiene un indicador del subtipo («`SYSIMG`»), de seis bytes, que indica la función a la que se ha destinado la partición. También se prevé la llegada de correcciones y extensiones al diseño inicial, de ahí la presencia

[6] F7h equivale a 100h menos 9. Se eligió este descriptor al no encontrarse ningún otro tipo de partición que lo use (al menos, ninguno soportado por las herramientas habituales).

de un número de versión de un byte (actualmente se usa la versión 0 del subtipo).

Además de la firma, la cabecera se compone de otros cinco campos. Los bits que no pertenecen a ninguno de ellos se consideran reservados (aquellos que no tienen ningún texto en la figura 3.3), aunque son respetados por la implementación de `f7disk`. De esos cinco campos, los cuatro primeros describen la configuración de la partición.

El primer campo, de cuatro bytes, indica el desplazamiento dentro de la partición donde comienza la primera ranura. El segundo, del mismo tamaño indica la longitud de las ranuras. Para permitir el alineamiento de las ranuras, uno de dos bytes representa la separación entre el final de una y el principio de la siguiente. El cuarto valor, de cuatro bits, representa el número de ranuras total menos uno: aprovechando que no tiene sentido una partición sin ranuras, se ha reservado todo el rango de representación para albergar valores válidos.

Finalmente, no todas las ranuras existentes en la partición tienen que estar en uso. El bitmap de dieciséis bits al final de la cabecera indica cuáles sí lo están, siendo el único campo que contiene información sobre el estado de la partición: mientras que los otros son inmutables, el bitmap es modificado cada vez que se instala o se libera una ranura.

3.3.2 Sobre «f7disk»

Como ya se ha introducido, `f7disk` es una utilidad que se ha creado para inicializar y gestionar particiones de tipo F7h. Está pensada específicamente para trabajar con unidades basadas en MBR. No se considera una herramienta de particionado per se, ya que su cometido no es asignar particiones en la tabla del registro de arranque. En su lugar, transforma otras ya existentes durante la inicialización.

Esto se ha hecho así para mantener las responsabilidades de cada utilidad separadas, permitiendo disponer de diferentes opciones en el particionado. Lógicamente, las herramientas clásicas no implementan el soporte que un nuevo tipo de partición necesita, pero sí respetan sus límites, por lo que son capaces de crear las particiones y gestionar el espa-

cio de almacenamiento restante de la unidad de arranque. Por ejemplo, las imágenes y unidades usadas en las pruebas fueron particionadas con `fdisk`.

Conviene recordar que el descriptor del tipo original de la partición es sobreescrito durante su transformación. Además, no es necesario haberle aplicado previamente ningún formato[7]. En este caso, las particiones eran asignadas automáticamente como de «tipo Linux»[8], evitando usar `mkfs.ext2` o similares en la destinada a albergar las imágenes del sistema.

Una vez que la unidad ha sido particionada y se ha decidido cuál de las particiones será transformada, es el momento de usar `f7disk`. La utilidad emplea una serie de subórdenes, siendo las más destacables:

- `tablebrief`
- `brief`
- `override`
- `clear`
- `load`
- `cpboot`
- `help`

Todas las subórdenes que reciben tamaños y desplazamientos como argumentos admiten su introducción en forma de unidades binarias. Al representar esa clase de valores en la salida, estos son agrupados empleando la unidad mayor con la que pueda realizarse una división entera exacta, algo que puede usarse como guía cuando no es posible una alineación perfecta de las estructuras de datos internas.

La primera suborden permite mostrar las cuatro particiones de la tabla en mayor detalle y sin importar si están desactivadas, para verificar la integridad de las entradas y averiguar cuál es el identificador numérico (del cero al tres) que se encuentra asociado a la partición a transformar.

[7] Algunas herramientas como GParted aúnan el particionado y la inicialización del formato en un solo paso. Esto no es un impedimento en la instalación, pero se desaconseja, ya que dicho formato será también sobreescrito.

[8] Descriptor 83h, utilizado por los formatos EXT2, EXT3 y EXT4.

El usuario usará ese identificador para «desautorizar» (suborden `override`) a la partición, cambiando su descriptor e inicializando la estructura de datos de la cabecera[9]. Aunque la herramienta intenta averiguar la configuración más adecuada, necesitará alguna indicación del usuario. Conviene usar el *flag* `-dry-run` mientras se deciden todos los ajustes para asegurar que ninguno es pasado por alto.

Cuando todo haya terminado, las herramientas clásicas indicarán que el tipo de la partición es desconocido, pero en `f7disk` será reconocido con la descripción «Mark 1». A partir de ahí, el usuario puede mostrar la configuración instalada (suborden `brief`), así como instalar imágenes en las ranuras o liberarlas.

Figura 3.4: Lectura de la configuración de una partición F7h.

```
% f7disk tablebrief mark1.img
# Boot Type      Start       Size        End Description
0  00h  F7h          5         64         68 Mark 1
1  00h  83h         69       2811       2879 Linux
2  00h  00h          0          0        N/A (disabled)
3  00h  00h          0          0        N/A (disabled)
% f7disk brief mark1.img 0
Active slots = 1/1
Bitmap = 0001
First = +1
Size = 31 KiB
Every = 31 KiB
%
```

Por conveniencia, también se ha incluído la suborden `cpboot` para instalar el cargador de arranque al inicio de la unidad. Como en el resto de órdenes, se comprueba que su ejecución respetará la posición y los límites de las particiones.

Finalmente, para liberar ranuras o instalar imágenes en ellas se han usado las subórdenes `clear` y `load`. Estas modifican el bitmap de la cabecera. Para minimizar el riesgo de corrupción en caso de fallo, se ha tenido en cuenta el orden en el que se vuelcan las imágenes y se conmutan

[9]Esto no es posible con particiones vacías. Tampoco con particiones protectivas GPT.

los bits de la cabecera. De esa manera se previene que una ranura con una imagen válida quede desactivada o que una imagen parcialmente copiada quede activa.

Capítulo 4

La imagen del núcleo

El núcleo es cargado en memoria sin que se haya abandonado el modo real. Eso significa que el núcleo reúne código escrito en lenguaje ensamblador y C, estando algunas partes destinadas al modo real de 16 bits y otras al modo protegido de 32 bits.

Establecer la distribución interna del cargador de arranque manualmente es razonable, pero los problemas de almacenamiento y uso de la memoria que el núcleo debe afrontar son de naturaleza muy diferente. Debido principalmente a las partes escritas en C, el tamaño de la imagen binaria crece durante el desarrollo con rapidez, y necesita cierta libertad para poder hacerlo. Esto puede lograrse delegando los detalles más nimios al enlazador.

Los desafíos vienen por dos frentes. Por un lado, el núcleo necesita saber en que zona de la memoria se encuentra todo lo que necesitará usar. De hecho, el enlazador debe conocer de alguna forma dónde se supone que se cargará el núcleo[1], para así trasladar las direcciones de memoria correctas al código máquina.

Por otro lado, la segunda etapa del cargador debe ceder el procesador saltando al código del núcleo. Para ello debe saltar a la dirección de memoria exacta donde se localiza el punto de entrada, pero debido a que

[1]O generar código máquina independiente de la posición (PIC). Pero esta opción conlleva sus desventajas, y dado que no se pretende que el núcleo se cargue en cualquier parte no parece demasiado sensato.

la generación del cargador de arranque no es asistida por el enlazador, no puede preguntarle por dicha dirección.

En este capítulo se detalla como está estructurada internamente la imagen del núcleo de Mark 1, así como la organización de su directorio de código fuente.

4.1 Combinación de C y NASM

El código para el modo protegido está escrito en C en su mayoría, aunque también hay algunos fragmentos en lenguaje ensamblador. En cuanto al código para el modo de 16 bits, se han escrito íntegramente en este último. Esto se debe a que esa parte no es extensa, y se ha considerado que combinar dos lenguajes en ella sería más perjudicial que beneficioso.

En total son tres tipos de código que han de ser ensamblados y enlazados en la misma imagen.

De vez en cuando, las rutinas de C deben ser invocadas desde el código de NASM (o viceversa). Es muy importante tener en cuenta que la convención de llamada predeterminada que aplica GCC es «**cdecl**», cuyas características más importantes son dos: que los argumentos son apilados de derecha a izquierda, y que el código invocador es responsable de restaurar la pila una vez que la función invocada ha retornado.

El primer aspecto es muy útil, porque permite a las funciones localizar al primer argumento que se les ha pasado incluso sin saber con exactitud cuántos han sido apilados. El segundo impacta en el tamaño de las imágenes porque exige códigos de operación extra con cada invocación, pero es bastante más robusto que delegar en la función invocada la limpieza de la pila.

En general, los valores retornados se transfieren a través del registro `EAX` (o `AX`/`AL`, según el tipo de dato). Además, GCC agrega una restricción notable que debe respetarse. Este compilador da por hecho que los registros `EBP`, `EBX`, `ESI` y `EDI` no son alterados entre invocaciones. Es por eso que las rutinas que hagan uso de ellos deben guardar sus valores originales y restaurarlos antes de retornar.

Tanto para invocar código de C desde NASM como al revés, la función invocada debe ser declarada como externa. En C, las funciones son externas de manera predeterminada, pero puede explicitarse mediante la palabra clave `extern`:

```
extern void asm_routine(type1 var1, type2 var2);
```

En NASM, la función es solamente un símbolo, incluso a nivel de la declaración:

```
[extern c_function]
```

4.2 Script del enlazador

El enlazador necesita algo de ayuda para generar la imagen correcta. Esta ha sido proporcionada a través de un script de enlazado[2] compatible con el *toolchain* de GNU que divide al binario en secciones:

- Sección de la función de entrada `.start`.
- Sección para el código basado en direcciones físicas.
- Sección general `.text`.
- Sección general `.rodata`.
- Sección general `.data`.
- Sección general `.bss`.

La sección `.bss` no se está usando todavía, pero es importante saber en qué consiste. Allí se sitúan los objetos cuyo tiempo de vida es estático y que no han sido inicializados explícitamente, o bien han sido inicializados a cero. Aunque hay formas de forzar su inclusión en la imagen, esta no se produce de manera predeterminada, ya que *se espera que el núcleo inicialice la memoria a cero durante el arranque* para ahorrar espacio. Pasar esto por alto es arriesgado, porque el núcleo accederá a regiones de memoria fuera de sus dominios, y leerá valores inesperados.

[2] Véase el fichero `/kernel/32/link.ld`.

Mientras que la mayoría de secciones son alineadas automáticamente por el enlazador, la primera se encuentra anclada a una dirección de memoria, lo que garantiza que el cargador pueda saltar de manera segura.

Las dos primeras secciones son diferentes al resto. Eso se debe a que están pensadas para trabajar con direcciones físicas de memoria, mientras que en el resto, la dirección de carga difiere de la dirección efectiva. Esto se explica en mayor profundidad en el capítulo dedicado a la memoria virtual.

4.3 Organización del código

Si el código del cargador está contenido en el directorio `/boot`, el del núcleo se localiza en `/kernel`. Dicho directorio esta estructurado de la siguiente forma:

- `/kernel/32`
- `/kernel/drivers`
- `/kernel/include`
- `/kernel/wrappers`

El código del primer directorio no se limita al del modo protegido. Independientemente de si contiene código de 16 bits, está pensado para máquinas de 32 bits, y por tanto se considera no portable. Su contraparte es el directorio `drivers`.

El directorio `wrappers` ('envoltorios') contiene rutinas escritas en lenguaje ensamblador que pueden invocarse como si se tratase de funciones escritas en C. El objetivo es que ciertas instrucciones que normalmente sólo son accesibles desde el código ensamblador puedan utilizarse también en ese lenguaje.

Las cabeceras utilizadas por el núcleo son organizadas en un árbol de ficheros bajo el directorio `include`. Por ejemplo, las interfaces de las funciones que se corresponden con los envoltorios se encuentran declaradas en el archivo `/kernel/include/asm.h`.

Finalmente, además del código específico del núcleo, debe recordarse que también se emplea una versión de la biblioteca libcmin, descrita en detalle en el capítulo 5.

4.4 Punto de entrada

El punto de entrada se encuentra en la rutina `_start`, de 16 bits, que se limita a redirigir al procesador a `entry_16`. Esta es responsable de conmutar al modo protegido, por lo que no es portable y está escrita en ensamblador. Su segunda parte, `entry_32`, realiza unos últimos ajustes y activa el mecanismo de paginación invocando a ciertas funciones escritas en C antes de hacer lo propio con `kmain`, la función principal del núcleo.

4.5 Drivers (controladores)

El código controlador permite interactuar con un dispositivo o protocolo determinado[3]. Para poder realizar algunas tareas básicas se han desarrollado controladores para el *framebuffer*, el cursor, el teclado, el temporizador principal, y los PIC.

Las particularidades de algunos de estos, y de otros fragmentos de código específico que se encuentran en el mismo directorio se explican individualmente en más detalle en el capítulo 6.

[3]Por extensión, no siempre se refiere al manejo de un componente físico (por ejemplo, un controlador EXT2).

Capítulo 5

Minibiblioteca de C

Como ya se adelantó en la sección 2.1, para que el núcleo sea construído, el *toolchain* de C puede necesitar un conjunto de funciones básicas. En realidad, su presencia también es conveniente para su uso desde el núcleo.

Este capítulo describe la biblioteca mínima de C que acompaña al código del núcleo de Mark 1, y que recoge la implementación de estas funciones.

Dado que los futuros programas de usuario también necesitarán una biblioteca de C, se ha sitúado en un directorio dedicado para poder compartir el código común: `/libcmin`[1]. Esto deja espacio para extenderla con el código específico de cada versión.

Puesto que las funciones del núcleo son enlazadas de manera distinta a como lo harán en la biblioteca de usuario, la versión de libcmin empleada en el núcleo es generada por el Makefile de este último.

5.1 Características y principios

Los tipos y funciones de la minibiblioteca que se basan en la biblioteca estándar de C completa son compatibles, y se ha resistido la tentación de probar algunos cambios. Esto ha sido necesario en el caso de las funciones potencialmente requeridas por el *toolchain*, pero además se cree que sería

[1] Es decir, un subconjunto mínimo de la biblioteca de C.

confuso proporcionar dos versiones sutilmente incompatibles de ciertos tipos, y de poca utilidad prescindir de las versiones estándar.

No obstante, también incluye extensiones que no tienen correspondencia en la biblioteca estándar de C. En la implementación actual, esto se refleja a través de la cabecera `u.h`[2], basada en la de Plan 9 y utilizándose de manera similar (como se describe en la página `intro(2)` del manual de Plan 9 [14]).

Esta `u.h` no proporciona todos los tipos definidos en la de Plan 9, y algunas de sus declaraciones se reemplazan por tipos de C99 y C11 con sus propias cabeceras. Se intenta que lo que queda declarado en `u.h` no se refleje en otras cabeceras para evitar esa dependencia fuerte que sí existe en Plan 9. A cambio, suele ser necesario agregar otras como `stddef.h` o `stdint.h`.

Puede resultar algo sorprendente la ausencia de guardas de cabecera en el repositorio. No es accidental, y también se sucede en Plan 9[3], donde *incluir una cabecera de manera indirecta está prohibido*. Es tarea de los desarrolladores realizar las inclusiones desde el fichero final, en el orden correcto y sin repetir ninguna.

Se ha seguido esta misma aproximación debido a que el mecanismo empleado por los compiladores de C y C++ para incorporar declaraciones de otros módulos no es demasiado sofisticado, y una organización de las cabeceras poco sensata puede generar problemas de compatibilidad, incrementar sustancialmente los tiempos de compilación y contaminar los ficheros de implementación con declaraciones inesperadas.

Las cabeceras «autocontenidas», de apariencia inicial más amigable, terminan causando estos problemas difíciles de razonar. En la creencia de que la única manera efectiva de combatirlos es reemplazar este sistema fuera del preprocesador[4], y que es preferible lidiar con errores de inclusión propios que los causados por otras personas en cabeceras de terceros, se

[2] El nombre de esta cabecera significa *unsigned*.

[3] Se aconseja consultar además el documento sobre el estilo de programación escrito por Rob Pike [15].

[4] Estos problemas influyeron en el diseño del lenguaje Go, historicamente relacionado con Plan 9. Por otro lado, se espera que C++20 los aborde próximamente mediante el nuevo sistema de módulos.

ha llegado a la conclusión de que la manera de proceder de Plan 9 es la solución de compromiso más razonable.

Es destacable que Plan 9 usa el tipo `long` directamente para representar tamaños, mientras que en Mark 1 se ha optado por `size_t`. A menudo se desaconseja el uso de tipos sin signo donde pretende usarse aritmética no modular, y aunque se prescribe extrema precaución, se ha considerado más conveniente utilizar este tipo todo el tiempo por consistencia [16]. Además, se sugiere que el problema no proviene del tipo en sí, sino que es consecuencia del tipado débil del lenguaje.

5.2 Requisitos del toolchain

La lista de funciones que debe proporcionar el desarrollador varía incluso entre versiones de un mismo *toolchain*, y suele incluirse en su manual [17]. En el caso de la versión utilizada de GCC, esas son `memcmp`, `memcpy`, `memmove`, y `memset`.

Algunas versiones antiguas pueden incluir otras como `bcopy` o `bzero`. Dado que implementar unas a partir de las otras es sencillo, se ha aprovechado para incluirlas también a través de la biblioteca de C de Mark 1 (véase `/libcmin`), en previsión de que los usuarios decidan utilizar otro *toolchain*. Sin embargo, se desaconseja utilizarlas directamente, y se advierte que son firmes candidatas a la eliminación en versiones futuras.

Las funciones `memcpy` y `memmove` son similares, pero la primera pretende ser más rápida (en lugar de resolver el problema general). Particularmente, `memcpy` debe evitarse si la memoria de destino puede solaparse con la copiada. La implementación de libcmin no realiza esta optimización por el momento, pero sí que se ha tenido en cuenta al usar ambas funciones.

5.3 Manejo de cadenas

Por pragmatismo, también se han implementado las funciones de manejo de cadenas. A diferencia de las basadas en el manejo de la memoria

en bruto, existen particularidades que derivan de la finalización de las cadenas en puntos de código nulos.

En la implementación concreta, se ha considerado que en el uso indefinido de las funciones que retornan un valor del tipo `size_t` se debe devolver el máximo. Aunque este valor no indica inmediatamente un error, es probable que lo sea, ya que el usuario no espera a menudo manejar cadenas tan amplias.

Capítulo 6

Modo protegido (32 bits)

En este capítulo se detalla el funcionamiento en modo protegido de la segmentación, las interrupciones, y la memoria virtual. También se describe cómo se gestionan estos aspectos en Mark 1, además de la implementación de un sencillo planificador de hilos del núcleo.

6.1 Introducción al modo protegido

Para hacer uso de algunas capacidades de la máquina, es necesario abandonar el modo real. La mayoría de procesadores modernos basados en la arquitectura x86 pueden trabajar en modo protegido de 32 bits. Algunas de sus características más destacables son las que siguen.

En primer lugar, el tamaño predeterminado de los operandos dobla al de las instrucciones en modo real. Ésto debe tenerse muy en cuenta al manipular la pila mediante instrucciones tales como `PUSH` y `POP`.

En segundo lugar, los registros extendidos de 32 bits pasan a ser los predeterminados en los códigos de operación. Los registros de datos de 16 bits, aún accesibles, se solapan con éstos constituyendo su parte baja. Los nombres de los registros extendidos son idénticos a los de aquellos, salvo que van precedidos por la letra «E» (`EAX`, `ECX`, `EBX`, etcétera). Además, se incorporan dos registros de segmento de propósito general adicionales: `FS` y `GS`.

En tercer lugar, en el modo real, las direcciones tienen un mínimo de 20

bits y los registros no tienen suficiente capacidad como para almacenarlas, así que el programador elige un rango de memoria direccionable 64 KiB (llamado segmento) en el que trabajar ayudándose de un segundo registro. Esta limitación es eliminada en el modo protegido, acompañada de un modelo de segmentación completamente distinto y bastante más complejo que debe configurarse primero, como se describe en la sección 6.2. Bajo este modelo, los desplazamientos de las direcciones de memoria pasan a ocupar cuatro bytes, aprovechando la capacidad adicional de los registros.

De forma similar, la IVT ('tabla del vector de interrupciones') preconfigurada del modo real es reemplazada por una **IDT** ('tabla de descriptores de interrupción'). Ésto se describe detalladamente en la sección 6.3.

En cuarto lugar, el modo protegido introduce el mecanismo de paginación que habilita el uso de una memoria virtual. Las particularidades de su configuración se describen en la sección 6.4.

Por último, cuando el salto de modo real a modo protegido concluye, conviene recordar que el procesador deja atrás el código de 16 bits y las rutinas del BIOS quedan inaccesibles. Eso significa que los desarrolladores no pueden dar por sentadas funcionalidades que en modo real se consideran básicas y que, en ocasiones, resultan difíciles de reemplazar.

6.2 Segmentación

Los sistemas operativos modernos no utilizan las características de la segmentación de los procesadores x86 y, a grandes rasgos, el modo de 64 bits no permite usarla. Aún así, debe ser «desactivada» configurando segmentos que cubran toda la memoria en la **GDT** (o 'tabla global de descriptores'). Este mecanismo es además necesario para poder conmutar al modo protegido de 32 bits.

Los descriptores se identifican por el desplazamiento en bytes en la tabla en la que se encuentran (denominado **segmento**), ocupando 8 bytes cada descriptor, y sabiendo que el primero de la tabla es nulo. El procesador usa ese identificador para acceder al descriptor, y usa la dirección base que éste último contiene para combinarla con el desplazamiento de cuatro bytes. Así es como los desplazamientos se convierten en **direccio-**

nes lineales. También significa que cada segmento puede direccionar un máximo de 4 GiB[1].

Además de eso, los segmentos pueden contener código o datos. Afortunadamente, también pueden solaparse convenientemente, así que para saltar al modo protegido y olvidarse de la segmentación pueden utilizarse dos entradas, ocupando cada una los primeros 4 GiB de memoria. En Mark 1 se usa la segunda entrada (8h) para el descriptor del segmento de código y la tercera (10h) para el de datos.

6.2.1 Descriptor de la GDT

El procesador accede a la GDT mediante un descriptor almacenado en el registro `GDTR` mostrado en la figura 6.1. Son relevantes las instrucciones de ensamblador `LGDT` (para cargar el descriptor desde la memoria) y `SGDT` (para guardarlo en memoria).

Figura 6.1: Registros de la tabla del sistema (sección 2.4 del manual de Intel [18]).

	47(79)	16 15 0
GDTR	32(64)-bit Linear Base Address	16-Bit Table Limit
IDTR	32(64)-bit Linear Base Address	16-Bit Table Limit

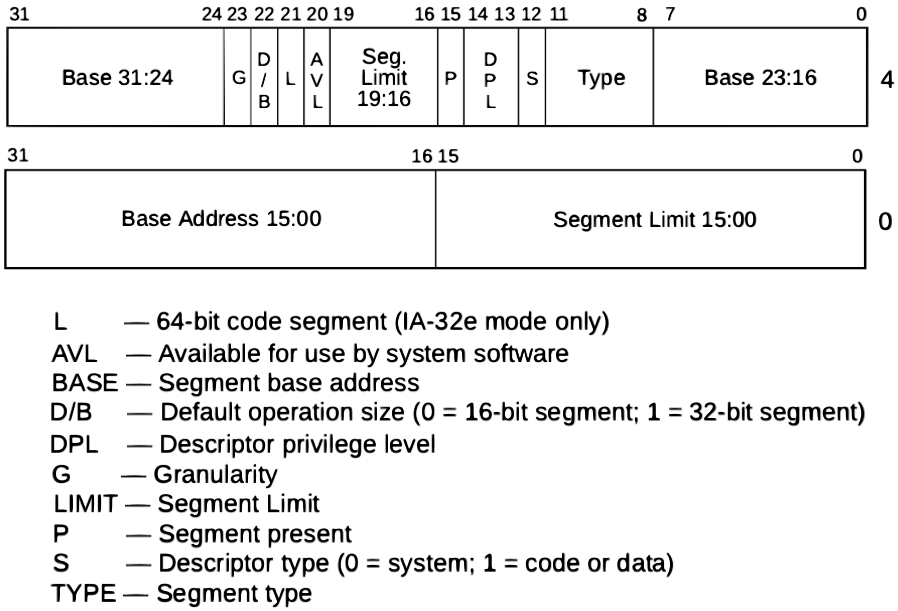
6.2.2 Descriptores de segmento

Mark 1 define sus descriptores de segmento (cuya estructura muestra la figura 6.2) en el fichero `/kernel/32/gdt.asm`. En el caso de los dos descritores destinados a la ejecución en modo protegido, se cubre toda la memoria y se activa el bit que determina el tamaño predeterminado de las operaciones (segmentos de 32 bits). La única diferencia entre ambos es el último bit que determina su tipo, y que indica si es un segmento de datos (0010b) o de código (1010b).

Tal y como ya se ha mencionado, Mark 1 puede regresar en ocasiones al modo real. Para ello, carga dos descriptores adicionales de 16 bits

[1]Sin ayuda de PAE (*Physical Address Extension*).

Figura 6.2: Descriptores de segmentos (sección 3.4.5 del manual de Intel [18]).



en los registros de segmento. Esto es necesario para preparar a la CPU antes de desactivar el modo protegido por completo. Mark 1 emplea los descriptores cuyos selectores son el 18h y el 20h para código y datos respectivamente.

Los cuatro descriptores están configurados para el nivel de privilegio más elevado (`DPL`), ya que por el momento solamente se trabaja en modo núcleo. También permanece activo el bit de granularidad (`G`), que hace que el límite se base en unidades de 4 KiB abarcando 4 GiB de memoria.

6.2.3 Conmutación al modo protegido

El proceso básico para conmutar al modo protegido consiste en desactivar las interrupciones (las no enmascarables pueden desactivarse mediante un puerto de entrada y salida del reloj de tiempo real del chip CMOS), cargar el registro `GDTR` mediante la instrucción `LGDT`, usar la instrucción `MOV` para activar el *flag* `PE` (*Protected Mode Enable*) del registro `CR0` y saltar (o invocar a una rutina) justo después de activar dicho *flag*.

En Mark 1, la implementación de la conmutación puede encontrarse

en el fichero `/kernel/32/entry.asm`. En teoría se puede configurar la paginación al mismo tiempo, pero se ha preferido inicializar las tablas trabajando ya en modo de 32 bits y usando código escrito en C:

```
entry_16:
        ...

        ; Se desactivan las NMI.
        in AL, 0x70
        or AL, 0x80
        out 0x70, AL
        ; Se enmascaran las demás interrupciones.
        cli

        lgdt [gdt_descriptor - correction]

        ; Se activa el bit PE.
        mov EAX, CR0
        or EAX, 0x1
        mov CR0, EAX

        jmp far [entry_32_addr - correction]
```

El uso de la macro `correction` se debe a que el salto desde el cargador de arranque al código de 16 bits del punto de entrada en Mark 1 se produce con un desplazamiento diferente al utilizado en modo protegido una vez cambia el modelo de segmentación.

6.3 Interrupciones

6.3.1 Consideraciones en modo real

En el arranque, una tabla de vectores de interrupción (o IVT) preinstalada en la dirección 0h de la memoria proporciona 256 **vectores de interrupción**, de cuatro bytes cada uno. En realidad, los «vectores de interrupción»

son las direcciones segmentadas de rutinas manejadoras.

El procesador usa el número de la operación de interrupción[2] como índice para calcular la dirección del manejador al que debe invocar. Además, los procesadores posteriores al 8088 preservan la dirección y el tamaño de la IVT en el registro `IDTR` (cuya estructura muestra la figura 6.1), lo que hace posible situarla bajo otras direcciones de memoria. Las direcciones de los propios vectores pueden calcularse sumando la dirección de dicho registro al índice de la tabla multiplicado por cuatro.

Los desarrolladores pueden usar esta información para reemplazar las rutinas manejadoras o crear sus propias tablas con cierta facilidad. Sin embargo, la IVT y los vectores de interrupción no sirven en modo protegido, y deben sustituirse por una IDT. Las dos tablas comparten el mismo registro aprovechando que no se pueden usar simultáneamente.

6.3.2 Gate descriptors

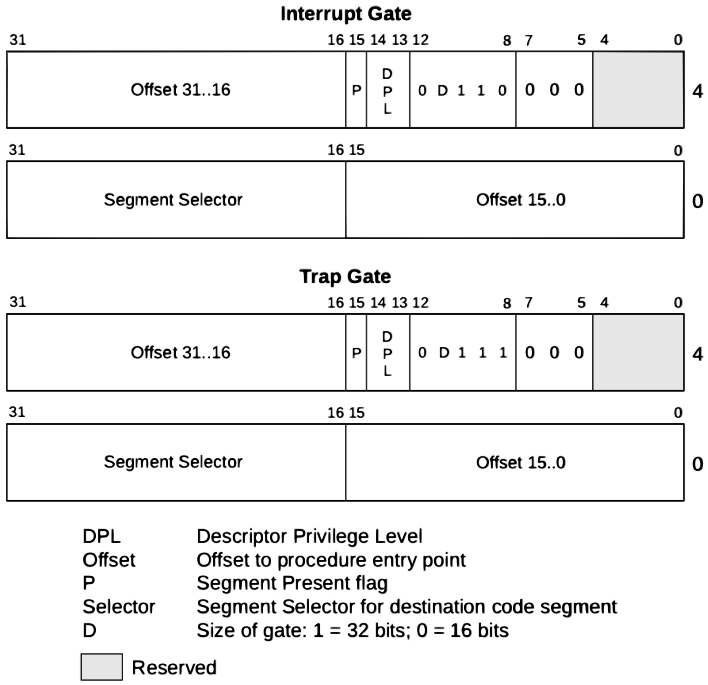
Los descriptores de la IDT se denominan ***gate descriptors***. Ocupan ocupan ocho bytes, y se clasifican en tres tipos: ***task***, ***trap*** e ***interrupt***. Mientras que los del primer tipo se utilizan para efectuar cambios de contexto vía *hardware*, los otros dos son relevantes en el reemplazo de las interrupciones en modo real. Tanto los *trap gates* como los *interrupt gates* comparten estructura (visiblemente más compleja que la de los vectores de interrupción) como muestra la figura 6.3, conteniendo un puntero a una **ISR** ('rutina de servicio de interrupción'). La única diferencia entre ambos es que, al usar un `interrupt gate`, el procesador bloquea subsiguientes interrupciones automáticamente mientras se ejecuta el manejador.

6.3.3 Implementación de las interrupciones en Mark 1

Mientras que otros sistemas operativos eliminan la IVT y las rutinas manejadoras a las que apunta, Mark 1 las preserva al principio de la memoria, inicializando una IDT en las secciones del núcleo. Así, al conmutar de

[2]Salvo para la instrucción `INT3`.

Figura 6.3: Gate descriptors para el manejo de interrupciones (sección 6.11 del manual de Intel [18]).



nuevo al modo real, sus interrupciones pueden restaurarse temporalmente volviendo a cargar el registro `IDTR`.

La IDT se configura programáticamente. Tanto su estructura como su descriptor se definen en el fichero `/kernel/32/idt.asm`, pero las entradas deben inicializarse invocando a la función `idt_reset`, definida en `/kernel/drivers/idt.c`. Para ello se vale de otra función del mismo fichero, `idt_install_isr`, desde el que se puede ver la configuración de los *trap gates* que Mark 1 utiliza.

Las rutinas de servicio de interrupción se dividen en dos partes: una función escrita en C y una rutina *wrapper* escrita en NASM que envuelve a dicha función[3]. Las direcciones de ambas están agrupadas formando sendas tablas (declaradas en el fichero `/kernel/32/isr.asm`), lo que sim-

[3]Nótese que estas rutinas invocan a código escrito en C, al contrario que con las definidas en el directorio `/kernel/wrappers`, que sirven para que el código en C invoque a instrucciones escritas en ensamblador.

plifica la implementación: el código en C usa la tabla de rutinas para cargar la IDT, y las rutinas en ensamblador apuntan a la función que realmente se encarga del manejo de la interrupción usando la otra tabla.

Aunque algunas interrupciones y *traps* reciben un código de error como argumento, no es así con todas. Para usar una interfaz común en las funciones de C y poder reutilizarlas fácilmente, se han usado dos tipos de *wrappers* generados mediante macros: `handler_simple` y `handler_with_errcode`. El primero se usa en aquellas en las que no existe código de error, que se sustituye apilando el valor 0. Las macros tienen por argumento el identificador de la interrupción:

```
...
handler_simple 0x07
handler_with_errcode 0x08
handler_simple 0x09
handler_with_errcode 0x0A
handler_with_errcode 0x0B
...
```

En los dos casos, se apila además el número que identifica a la interrupción, y se salta a la rutina `handler_common`. Con la finalidad de ahorrar espacio, ésta realiza aquellas operaciones necesarias en todos los manejadores: preparar la pila, acceder a la tabla de funciones usando el identificador previamente almacenado y limpiarlo todo antes de retornar de la interrupción.

Las funciones manejadoras reciben también como argumentos separados el estado de cada registro relevante del procesador. Esto es así para evitar el uso de extensiones en C para garantizar el alineamiento de una sola estructura. La macro del fichero de implementación `/kernel/drivers/isr.h` simplifica la declaración y definición de estas funciones. Un ejemplo de implementación es el manejador predeterminado, definido en `/kernel/32/unhandled_interrupt.c`. En caso de error, este se disparará mostrando el contenido de los registros antes de detener la ejecución del núcleo invocando a la función `halt`.

Finalmente, las interrupciones son señalizadas a través de dieciséis

pines pertenecientes a dos PIC ('controladores de interrupciones programables'). Nótese que los pines `IRQ0` a `IRQ7` están vinculados a las interrupciones que van de la 08h a 0Fh, que se solapan con los identificadores de ciertas excepciones. Para evitar colisiones, los PIC se reconfiguran a través de las funciones definidas en `/kernel/drivers/pic.c`, de manera que todos los pines IRQ queden vinculados a interrupciones libres.

6.3.4 Temporizador (driver)

El *hardware* de los compatibles incluye un **chip PIT** ('temporizador programable de intervalos') 8253 u 8254 provisto de un oscilador y tres divisores de frecuencia, contando cada uno con un canal de salida. En realidad, sólo el primero de ellos está conectado al PIC, generando interrupciones a través del pin `IRQ0`. El segundo se usaba antiguamente para mantener el ciclo de refresco de la DRAM cada 15 ms aproximadamente, estando su uso reservado. En cuanto al tercero, era el que generaba la señal del altavoz integrado. Las garantías sobre el uso de estos dos últimos canales en máquinas modernas son escasas, pero no es así con el primero, que sigue siendo de gran utilidad. Aunque el PIT no se caracteriza por su precisión[4], pero suele servir para implementar la multitarea.

En Mark 1 se ha creado una versión primigenia del controlador para configurar el primer canal del PIT y generar una señal de onda cuadrada (correspondiente con el modo de funcionamiento `011b`). El código que se encarga de ello puede hallarse en el fichero `/kernel/drivers/timer.c`. Para ello, se usa el envoltorio de la instrucción `OUT`, manipulando el puerto de datos 40h (perteneciente al primer canal). y el puerto de control 43h del PIT. Tras la reconfiguración del PIC, el pin `IRQ0` se corresponde con la interrupción `20h`. La función escrita en C del manejador se encuentra en el mismo fichero, y se encuentra instalada en la tabla de funciones del fichero `/kernel/32/isr.asm`.

A diferencia del manejador predeterminado, en este caso se espera que

[4]Para lo cual puede recurrirse al reloj de tiempo real CMOS, asociado al pin `IRQ8`, o bien a los HPET ('temporizadores de eventos de alta precisión'), introducidos para reemplazar a estos dos [19].

la interrupción se dispare siempre a través de los PIC, y es por eso que tanto la función que configura el PIT como la función del manejador se sirvan del *driver* de éstos. Por un lado, la configuración concluye desenmascarando la `IRQ0` vía `pic_enable_irq`. Por otro, una vez que la función del manejador concluye con su cometido, realiza una confirmación antes de retornar invocando a `pic_ack_irq`. Los identificadores del pin de cada IRQ tienen unas macros correspondientes en la cabecera de implementación `/kernel/drivers/pic_macros.h` que se utilizan en ambas funciones de controlador del PIC. En este caso, el manejador se corresponde con la macro `IRQ_SYSTIMER`.

6.4 Gestión de memoria

Para poder usar la memoria que se encuentra más allá del primer MiB, no basta con habilitar la línea de direccionamiento A20. Por un lado, no se pueden hacer muchas suposiciones sobre la cantidad total de memoria disponible y, por otro, el riesgo de que algunos rangos de direcciones estén siendo utilizados para el *mapeo* de hardware no puede ignorarse. En tales casos, los rangos reservados varían ente máquinas y fabricantes. Es por eso que es necesario detectar qué regiones están libres para ser utilizadas.

Resulta obvio que permitir el acceso directo a toda la memoria direccionable es inseguro. El uso de una memoria virtual consiste en convertir a las direcciones de memoria lineales en otras llamadas **direcciones físicas**[5]. Esto se logra mediante el **mecanismo de paginación**, y con ayuda de un componente físico: la MMU (la 'unidad de gestión de memoria'). Además, permite subdividir la memoria en **páginas** con su propia configuración y permisos de acceso.

De manera predeterminada, la paginación se encuentra deshabilitada. Eso significa que las direcciones de memoria lineales se corresponden con direcciones físicas idénticas. A no ser que el código máquina generado no

[5]Las direcciones de memoria en su forma previa a la traducción del segmento suelen conocerse bajo esta configuración como **direcciones virtuales**, distinguiéndolas a su vez de las direcciones lineales.

sea independiente de posición, no se podrá invocar a las mismas funciones antes y después de activarla. De hecho, la rutina que se encargue de conmutar al procesador deberá escribirse con especial cuidado, ya que habrá de trabajar con ambas configuraciones.

Las estructuras de datos, el proceso de activación y las diferentes configuraciones para el mecanismo de paginación están documentadas en el capítulo 4 del manual del desarrollador de Intel [18].

6.4.1 Detección de la memoria

Para simplificar el uso de la memoria, Mark 1 realiza ciertas suposiciones conservadoras. En primer lugar, es una precondición que el sistema tenga más de 1 MiB de memoria instalada. El primer MiB está reservado para contener a la imagen del sistema, la pila del núcleo y algunas de sus estructuras básicas. Teniendo en cuenta que los programas más básicos trabajan en esta región de la memoria y que sus zonas seguras están razonablemente estandarizadas (según se explica en el capítulo 3), no se realiza ninguna comprobación adicional. Aunque eso permitiría aprovechar hasta un máximo de 127 KiB adicionales bajo la EBDA en algunas máquinas tal y como muestra la figura 3.1, se ha considerado que no compensaría las potenciales diferencias de comportamiento entre máquinas ni la complejidad adicional que supondría configurar la pila, situada en esa zona (lo más lejos posible de la imagen).

Para usar el resto de la memoria, la detección necesaria es realizada por el núcleo antes de conmutar al modo protegido (`/kernel/32/entry.asm`) consultando a una rutina del BIOS usada a través de la interrupción `15h` y conteniendo el valor `E820h` en el registro `EAX`, tal y como describe la especificación ACPI [20]. La rutina del núcleo que realiza la consulta itera hasta rellenar una tabla, ambas definidas en el fichero `/kernel/32/smap.asm`. Cada iteración genera un valor de 20 o 24 bytes en función de si se incluyen unos atributos extendidos de ACPI 3.0 (para los que se ha reservado espacio, pero que son ignorados en cualquier caso) que representa a un rango o región de la memoria. La tabla tiene un tamaño fijo, pero puede modificarse si fuese necesario.

Figura 6.4: Estructura de la interfaz Query System Map Address (sección 14.1 de la especificación ACPI, revisión 3.0b [20]). Cuando los atributos extendidos están presentes estos ocupan cuatro bytes más.

Offset in Bytes	Name	Description
0	BaseAddrLow	Low 32 Bits of Base Address
4	BaseAddrHigh	High 32 Bits of Base Address
8	LengthLow	Low 32 Bits of Length in Bytes
12	LengthHigh	High 32 Bits of Length in Bytes
16	Type	Address type of this range
20	Extended Attributes	See Table 14-5

Los bytes que indican el tipo de la región pueden adquirir diferentes valores, pero todos salvo aquellos de tipo 1 (memoria libre) son tratados automáticamente como reservados, e ignorados como cualquier región que no aparezca en la tabla.

Para finalizar, el código del modo protegido definido en el fichero `/kernel/drivers/paging.c`. accede a la tabla para hacer algunos ajustes finales y consultar las zonas libres de la memoria.

6.4.2 Memoria virtual

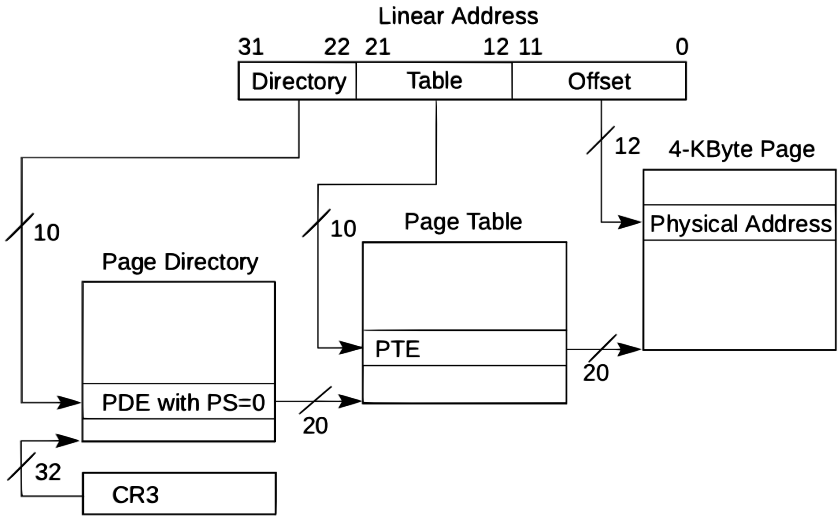
La memoria física puede considerarse dividida en **marcos de página** siendo el equivalente de las páginas en memoria virtual. Tanto los marcos como las páginas están siempre alineados en función de su tamaño, que puede ser 4 KiB o 4 MiB[6]. Eso significa que pueden direccionarse eliminando los últimos 12 y 22 bits de las direcciones de memoria respectivamente.

La configuración de la memoria virtual permite usar un **directorio de páginas** para asociar a las direcciones pertenecientes a cada página con las de un marco físico.

La estructura de datos principal es la **tabla del directorio de página** o PDT, que ocupa un marco de 4 KiB. Cada una de sus 1024 entradas

[6]Aquellas páginas cuyo tamaño supera el estándar de 4 KiB también son denominadas *huge pages* en Linux, *super pages* en BSD y *large pages* en Windows. Los tamaños disponibles varían con la arquitectura y la configuración.

Figura 6.5: Traducción de direciones lineales a páginas de 4 KiB usando paginación de 32 bits (sección 4.3 del manual de Intel [18]). Cada tabla del directorio multinivel puede direccionarse usando diez bits.



contiene un puntero físico que puede hacer referencia directa a un marco de 4 MiB, o bien a una **tabla de página** (PT) en un marco de 4 KiB, pudiendo cubrir un máximo de 4 GiB de memoria[7]. En el primer caso, una página de 4 MiB identificada por los primeros 10 bits de su dirección es asociada al marco apuntado. En el segundo, todas las páginas de 4 KiB identificadas por una dirección cuyos primeros 10 bits se correspondan con la entrada de la PDT serán asociadas a 1024 marcos del mismo tamaño, siendo sus direcciones físicas indicadas por las entradas de la PT subyacente. El resto de bits son empleados como desplazamiento dentro del marco físico.

Mark 1 usa tablas de página en todas las entradas de la tabla del directorio, como muestra la figura 6.5. Al igual que con la IDT, las estructuras de datos del directorio de páginas también se inicializan programáticamente, tal y como puede verse en el fichero `/kernel/paging.c`. Los marcos de página seleccionados para albergarla se encuentran en la

[7]Siempre asumiendo que el procesador está en modo de 32 bits y que PAE permanece desactivada.

memoria libre que se encuentra sobre el primer MiB, justo después de otra estructura: el bitmap de la memoria.

El bitmap ocupa 64 marcos, y mantiene dos bits por página. El objetivo es identificar que páginas están siendo utilizadas y cuales están disponibles para poder asignar su uso al núcleo o a otros procesos.

Tras inicializar sendas estructuras, la función de inicialización carga la dirección del directorio de páginas en el registro `CR3` y procede a activar el bit `PG` (*Paging Enable*) del registro `CR0` con ayuda del código definido en el fichero `/kernel/32/enable_paging.asm`:

```
enable_paging:
        ...

        mov EAX, [ESP + 4]
        mov CR3, EAX

        mov EAX, CR0
        or EAX, 1 << 31
        mov CR0, EAX

        ret
```

Al igual que otros núcleos en modo de 32 bits (por ejemplo, Linux en su configuración estándar [21]), Mark 1 está pensado para ejecutarse a partir de la dirección virtual `C0000000h`. El propósito es poder crear una versión de la tabla para procesos en la que los tres primeros GiB de memoria virtual sean ocupados su propia imagen, y todos compartan en el último la del núcleo.

Hasta este punto, la configuración ha sido realizada por un código separado enlazado para ejecutarse al principio de la memoria direccionable, y es necesario saltar a las funciones del resto del núcleo. Es por ello antes de activar la paginación se prepara ese salto manipulando la dirección de retorno desde el código ensamblador:

```
[extern kernel_area_vaddr_]
```

```
...

enable_paging:
        ; La dirección de retorno es apilada durante la llamada.
        mov EAX, [ESP + 8]
        or EAX, kernel_area_vaddr_
        mov [ESP + 8], EAX

        ...
```

6.5 Planificación básica de hilos en Mark 1

Mark 1 implementa un sencillo planificador expropiativo para hilos del kernel. La implementación prescinde de la conmutación vía *task gates* (mencionados en la sección 6.3) y se basa en el controlador del temporizador descrito en la sección 6.3.4 junto con algunas estructuras de datos adicionales.

6.5.1 Task State Segment (TSS)

La arquitectura del procesador define una estructura de datos llamada **TSS** ('segmento de estado de tareas') que contiene los valores de algunos de los registros del microprocesador. Sus campos y distribución se muestran en la figura 6.7.

Los campos `Previous Task Link` y `LDT Segment Selector` pertenecen a características que no se están usando[8], así que Mark 1 no los tiene en cuenta.

En Mark 1 se define una de estas estructuras por tarea, teniendo 104 bytes de longitud cada una. Para evitar tener que manipular el alineamiento de los *struct* de C mediante extensiones no portables, se ha creado una interfaz con el fin de poder acceder a esos datos:

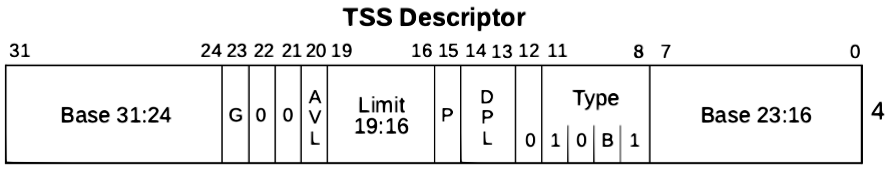
[8]La conmutación mediante *task gates* y las **LDT** ('tabla de descriptores locales') respectivamente.

```
typedef struct {
        uint prev_task;
        uint ss0;
        uint ss1;
        uint ss2;
        uint_fast32_t esp0;
        uint_fast32_t esp1;
        uint_fast32_t esp2;
        uint_fast32_t cr3;
        uint_fast32_t eip, eflags;
        uint_fast32_t eax, ecx, edx, ebx;
        uint_fast32_t esp, ebp;
        uint_fast32_t esi, edi;
        uint es, cs, ss, ds, fs, gs;
        uint ldt_segment_selector;
        uint io_map_base_addr;
} TSS;

TSS read_raw_tss(void *raw);
write_raw_tss(void *raw, TSS *tss);
```

Cada TSS permanece la mayor parte del tiempo en su forma binaria para poder ser manipulada por el microprocesador automáticamente cuando la tarea detiene su ejecución.

Figura 6.6: Parte alta del descriptor TSS (sección 7.2.5 del manual de Intel [18]).

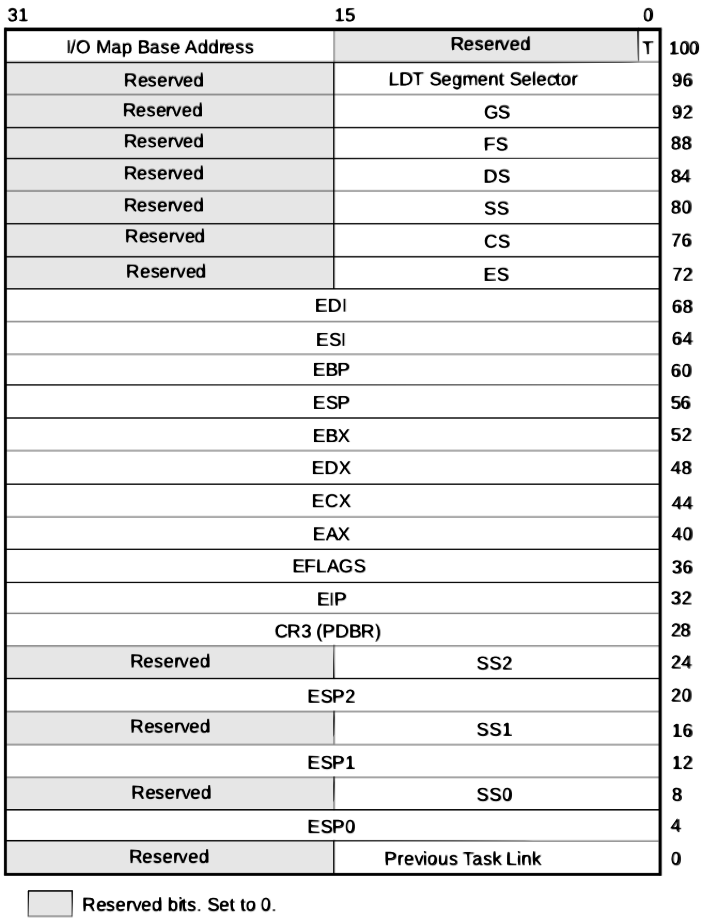


La TSS debe instalarse en forma de segmento empleando un descriptor de la GDT. Mark 1 reserva a tal fin el descriptor identificado por el selector 28h, siendo necesario cargar dicho valor en la parte visible del registro `TR`

(mediante la instrucción `LTR`). De manera similar a lo mostrado con otros registros, se proporciona una rutina a modo de *wrapper* llamada `load_tr`.

Esto solamente es necesario realizarlo la primera vez, en parte debido a que el procesador actualiza el registro por sí mismo durante los cambios de tarea, pero especialmente porque se ha empleado un único descriptor de la GDT para todas ellas. Esto significa que en lugar de cambiar de selector, se modifica el descriptor para que apunte a una TSS distinta (esta vez usando la rutina `load_tss`).

Figura 6.7: Task State Segment de 32 bits (sección 7.2.1 del manual de Intel [18]).



6.5.2 Bloque de control de tarea

La estructura del bloque de control de tarea de Mark 1 es bastante simple:

```
#define TSS_Size 104
typedef struct TaskBlock TaskBlock;

enum TaskStatus {
        TaskEmpty = 0,
        TaskReady = 1,
        TaskRunning = 2,
        TaskBlocked = 3,
};

struct TaskBlock {
        TaskStatus status;
        int next_pid;
        uchar tss[TSS_Size];
};
```

El tipo enumerado indica el estado actual de la tarea. El último estado definido (`TaskBlocked`) no está todavía en uso y se reserva para implementar las esperas por entrada y salida. Cada bloque contiene su segmento de estado, que ya contiene todos los registros que Mark 1 necesita[9].

Las estructuras de tipo `TaskBlock` son instanciadas en un array formando una tabla. Su índice se corresponde con el **PID** (o 'identificador de proceso'). Además, todas las entradas se encuentran encadenadas mediante sus PID formando una o dos listas circulares:

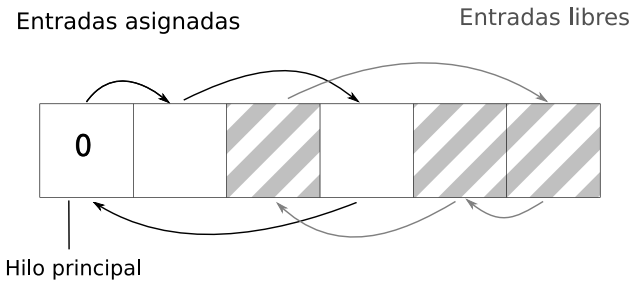
```
volatile int running_pid;
volatile TaskBlock *empty_tasks;
```

La variable `running_pid`, siempre contiene el índice de la entrada que pertenece a la tarea en ejecución. El resto de entradas en su lista circular son de tareas listas para continuar (o que permanecen bloqueadas). El

[9]Si se agregasen nuevos registros se situarían directamente en la estructura.

puntero, `empty_tasks`, hace refencia a aquellas entradas que no están asignadas a ninguna tarea (`TaskEmpty`). En caso de que todas las entradas estén ocupadas, este puntero contendrá el valor nulo.

Figura 6.8: Diagrama de las entradas de las tareas.



Además, siempre existe una tarea en ejecución, ya que el hilo principal del núcleo, con el PID 0 reservado (único PID no positivo) debe estar presente en todo momento. Para garantizar que esto se cumple, solamente los estados `TaskReady` y `TaskRunning` se permiten en el hilo principal.

Finalmente, las funciones `task_alloc` y `task_free` permiten inicializar una entrada vacía para crear un nuevo hilo o terminar con la ejecución del actual respectivamente.

6.5.3 Planificador

El planificador es invocado a través de la función manejadora definida en el fichero `/kernel/drivers/timer.c`:

```
ISR(timer_tick) {
        ++timer;
        ...
        pic_ack_irq(IRQ_SYSTIMER);
        kschedule();
}
```

La función `kschedule` implementa el algoritmo de planificación *round-robin* aprovechando la lista circular formada por las entradas en uso de la tabla. Para ello accede al apuntador de la entrada de la tarea en ejecución,

cargando su TSS y actualizando el PID almacenado.

```
next_pid = entries[running_pid].next;

tasks[running_pid].status = TaskReady;
tasks[next_pid].status = TaskRunning;
running_pid = next_pid;

load_tts(&tasks[running_pid].tss);
switch_task();
```

Para finalizar, la rutina `switch_task` envuelve a una instrucción de salto en ensamblador que fuerza el cambio de tarea usando el selector de la GDT anteriormente mencionado:

```
switch_task:
        jmp far 0x28:0h
```

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

Este trabajo sienta las bases de Mark 1 con el objetivo de hacerlo funcionar sobre la arquitectura de los compatibles (x86) de 32 bits. Para ello, se han desarrollado cuatro módulos principales: el cargador de arranque, la utilidad `f7disk`, la minibiblioteca de C y el núcleo.

Los dos primeros, junto con el diseño de un nuevo tipo de partición, atienden en su totalidad a necesidades introducidas por la compleja secuencia de arranque de la arquitectura, limitada a la carga automática de un programa mínimo para su ejecución en modo real, y que no ofrece un procedimiento estandarizado para cargar al sistema operativo completo.

La aproximación seguida ha permitido crear un cargador de arranque más simple rompiendo la dependencia circular entre el sistema de ficheros y el propio sistema operativo. Asimismo, el diseño pretende animar a los usuarios a experimentar modificando el sistema, apostando por los reinicios «suaves» para agilizar las pruebas y la capacidad de almacenar múltiples imágenes del núcleo en la misma unidad de arranque, a costa de incrementar ligeramente la complejidad de la solución.

Un aspecto clave es que las imágenes del núcleo quedan aisladas del resto del espacio de almacenamiento, de manera que la instalación del sistema operativo no impide a los usuarios continuar gestionando sus unidades mediante herramientas ya existentes. Esto hace posible iniciali-

zar sistemas de ficheros en particiones contiguas, así como instalar otros sistemas operativos.

Tras llevar esta solución a la práctica y haberla probado con Mark 1, se observa además que tanto el nuevo tipo de partición como la utilidad desarrollada son lo suficientemente generales como para ser reutilizados por otros núcleos.

La biblioteca mínima de C es necesaria para satisfacer los requisitos del *toolchain* que construye las imágenes del sistema. Su código y el del núcleo están separados con el propósito de generar dos versiones: una para el propio núcleo y otra para los programas de usuario. Además se han proporcionado algunas declaraciones semejantes a las de la cabecera `u.h` de la biblioteca de Plan 9 con el fin de facilitar la escritura de código.

El último módulo es el núcleo de Mark 1, que completa el proceso de arranque conmutando al modo protegido de 32 bits, configurando la memoria virtual, instalando las interrupciones en una IDT y habilitando la ejecución de hilos en modo privilegiado a través de un sencillo planificador expropiativo. También incluye controladores para el *framebuffer*, el cursor, el teclado y el temporizador del chip PIT ('temporizador programable de intervalos'). El código de este módulo es el más heterogéneo al estar compuesto de fragmentos escritos en C y en ensamblador, abarcar los modos real y protegido y tener que funcionar antes y después de la activación de la memoria virtual. Es también el más complejo, ya que muchas de las características relacionadas con la configuración de la arquitectura muestran fuertes interdependencias, y por ende son propensas a afectar a otras partes del sistema. La manifestación más notable de ello se dió en el código que se encarga de la memoria virtual, ya que todas las funciones y estructuras previamente desarrolladas fueron desplazadas a nuevas direcciones de memoria. Esto hizo necesario aplicar cambios a la tabla de descriptores de interrupción y extender la configuración de la segmentación.

Finalmente, uno de los mayores desafíos durante el desarrollo ha sido la depuración del código, debido a las restricciones del entorno de ejecución y a las limitaciones del propio Mark 1. La mayor parte del proceso ha consistido en simular la presencia de puntos de interrupción para exa-

minar el contenido de los registros y la memoria en bruto. La posterior incorporación de funciones de impresión al controlador del *framebuffer* hizo que la detección de errores fuese más fácil. Sin embargo, algunos de ellos solamente pudieron detectarse gracias a la realización de pruebas en entornos de ejecución diferentes, subrayando la importancia de no ceñirse al mismo emulador utilizado durante la implementación de las características.

Todos los módulos desarrollados han sido publicados amparados por una licencia de código libre y abierto, en sintonía con su propósito didáctico. Estos están distribuidos en dos repositorios: uno para Mark 1 y otro para la utilidad `f7disk`.

- `https://github.com/mikelcaz/Mark-1`
- `https://github.com/mikelcaz/f7disk`

7.2 Trabajo futuro

A partir de las bases que este proyecto ha planteado, se sugieren diferentes vías de desarrollo futuro.

En primer lugar, es deseable que el núcleo actual incorpore nuevas características para complementar a las ya implementadas. Por ejemplo, un controlador para el puerto serie habilitaría el uso de un depurador en los entornos emulados. El cargador aún debe implementar la carga multipista y dar soporte completo a VirtualBox, e integrar el soporte de teclado en modo real para convertirse en un monitor de arranque interactivo. También es conveniente implementar el soporte a procesos de usuario y completar el reinicio suave en el lado del núcleo.

En la misma línea, también se espera la implementación del sistema de ficheros. Se aconseja basar su implementación en el protocolo **9P2000** básico de Plan 9 [22], y después considerar pequeñas modificaciones con las que mejorarlo [23].

En segundo lugar, se ha tenido en cuenta en la estructura de directorios la posible reimplementación bajo nuevas arquitecturas. La sección 2.1

indica concretamente cómo instalar el *toolchain* para la variante de 64 bits de la arquitectura usada en el proyecto y, dado que ambas comparten muchas similitudes, darle igualmente soporte es el siguiente paso lógico.

En tercer lugar, la partición F7h puede ser mejorada para incluir el espacio ocupado de las ranuras, o bien aprovechar el espacio disponible en el sector de la cabecera para usarlo como sector de arranque. La estandarización en este sentido contribuiría a su reutilización por parte de otros sistemas operativos compatibles.

Por último, se anima a la creación de modificaciones y nuevas implementaciones [1]. Se sugiere el uso de lenguajes como Go, Rust u otros orientados a la programación de software de sistema. También el uso de otros ensambladores, ya sea explorando la rama de AT&T con el de GNU o la famila de los de tipo Intel con FASM.

Apéndice A

Introduction

The main general purpose operating systems are complex and full of functionality. Even though their designs have been polished over time, they have also been carrying an historical baggage which can be sometimes mistaken with an essential or deliberated part of them[1]. Lastly, the continuous incorporation of new abstraction layers makes reason about their inner workings an increasingly difficult task.

In this Master Thesis the first steps are taken in order to develop a free and open operating system with didactic purposes. To this effect, it is intended to use a design as simple as possible, mantain a restricted set of features and reduce those historical distractions as much as posible. The results are expected to encourage to other students to assemble new operating systems [1].

This project is inspired by others which share that didactic spirit. Many of them are based on UNIX and, similarly, this one adopts some traits of **Plan 9**[2] [2], motivated by the observation that its design is useful to illustrate the abstractions of the conventional operating systems [3] and being an opportunity for those students who never heard about it to discover it.

It must be noted that this memory does not provide a thorough explanation about the design and functionality of an operating system. Other

[1] Their file systems show clear examples of it, such as the heterogeneous mixtures of syntaxes in MS-DOS or the contentional usage of the `/usr` (üser") directory in UNIX-like systems.

[2] Originally known as *Plan 9 from Bell Labs*.

sources are referenced for that matter [4]. The memory describes the currently developed components of this operating system, explaining the most important concepts and techniques used in the process. It also contains some of the design decisions taken and how the main challenges has been faced during project implementation.

Project objectives

The main objective of this project is laying the foundations of the new operating system for compatible computers (x86), which has been named **Mark 1**. All the code must be made available for the comunity to be used as reference, especially in the academic field. It must also be easy to modify, build and run on an affordable basic environment.

It is important that Mark 1 be able to run on a virtual machine, in order to make it easy to follow the necessary instructions to modify and test the system. The chosen virtualization software has been VMware Workstation Player 15, although it is expected to extend the support to VirtualBox soon.

The first version of Mark 1 developed in this project must fulfill the next requirements:

- Provide basic support for the x86 architecture.
- Be able to load itself in memory from an external drive.
- Switch to 32-bit protected mode.
- Establish a minimum input/output system.
- Set up a basic virtual memory configuration.
- Run several kernel threads.
- Provide a C minilibrary usable by the kernel.

Work plan

The project consisted on the design and implementation of features belonging to four big modules: the bootloader, the C minilibrary, the kernel and

the `f7disk` utility. These has been incorporated through weekly iterations, subdivided in phases:

1. Study of the material related to the features.
2. Design of a prototype.
3. Implementation and experimentation.
4. Debugging of the prototype.

In the kernel module, some implemented features are strongly related (for instance, both the virtual memory and the advanced interrupt handling relies on enabling the protected mode) so its progressive completion unlocks new work lines. Due to that, iterations ended revaluating the priority of pending features and proving which new ones were practicable based on the progress done the previous week.

Memory structure

- Chapter 3 details the relevant tools included by the **development environment**, listing the reasons why they have been chosen and presenting the used configuration.
- Chapter 4 describes the bootstrap focusing on the design of two componets: the **bootloader** and the `f7disk` **utility**.
- Chapter 5 follows presenting the **kernel structure**, both at binary and source levels. For the first time, assembly code an C code are combined, the concept of driver is introduced and the base is laid for the rest of the components.
- Chapter 6 looks into the **C minilibrary**. Its code is split from the code of the kernel to reuse it with ease.
- Chapter 7 presents **interrupt handling**, **virtual memory** y **threads scheduling**.
- Chapter 8 and Chapter 9 expose the **final conclusions**, and show potential **future work** lines.

Apéndice B

Conclusions and future work

B.1 Conclusions

This work lays the foundation of Mark 1, being the x86 architecture of compatible computers (in its 32-bit variant) the target. To this effect, four main modules have been developed: the bootloader, the `f7disk` utility, the C minilibrary and the kernel.

The first two modules, along with the design of a new partition type, are designed as a whole to meet the needs introduced by the complex bootstrap phase of the architecture, limited to the automatic load of a minumum program intended to run on real mode, and which not offers a standard procedure to load the complete operating system.

The followed approach has allowed to create a simpler bootloader by the means of breaking the circular dependency between the file system and the operating system itself. Also, the design pretends to encourage the users to experiment modifying the system, using soft resets to speed up the testing and the capacity of storing several kernel images in the same boot drive, at the expense of a slightly increse of the solution's complexity.

A key aspect is how the kernel images are isolated from the remaining storage space, so the instalation of the operating system does not prevent the users from using existing tools to keep managing their drives. This makes possible to initialize file systems in adjacent particions, as well as install other operating systems.

After putting this solution into practice and having tried it with Mark 1, it is also noticed how both, the new partition type and the developed utility, are sufficiently general to be reused by other kernels.

The minimum C library is necessary to satisfy the requirements of the toolchain which builds the system images. Its code and the code of the kernel are split for the purpose of generating two binary versions: one for the kernel itself and other for the user programs. Moreover, some declarations have been provided in a similar fashion of the declarations of the `u.h` header from the Plan 9's library in order to write the code with ease.

The last module is the Mark 1 kernel, which completes the bootstrap switching to the 32-bit protected mode, setting up the virtual memory, installing the interrupt routines in a IDT and enabling privileged mode threads to run with the help of a simple preemptive scheduler. It also includes drivers for the framebuffer, the cursor, the keyboard and timer of the PIT chip (*Programmable Interval Timer*). The code of this module is the most heterogeneous as is made of parts written in C and in assembly, comprises both real and protected modes and has to work before and after enabling the virtual memory. It is also the most complex, as many of the features related to the setup of the architecture show strong interdependencies, and they are therefore more prone to affect to other parts of the system. The most remarkable manifestation of that happened when writing the code related to the virtual memory, as every function and structure previouly developed were shifted to new memory addresses. This made necessary to apply changes to the interrupt descriptor table and to extend the segmentation setup.

Finally, one of the biggest challenges during the development was debugging the code, due to the restrictions of the execution environment and the limitations of Mark 1 itself. Most of the process consisted of simulating the presence of breakpoints to examine the contents of the registers and the raw memory. The later incorporation of printing functions into the framebuffer driver made the detection of errors easier. However, some of them were detected thanks to the execution of tests over different enviroments, highlighting the importance of testing outside the emulator

used during the implementation of the features.

All developed modules have been published under a free and open source license, into line with its didactic purpose. They are distributed in two repositories: one for Mark 1 and other for the `f7disk` utility.

- https://github.com/mikelcaz/Mark-1
- https://github.com/mikelcaz/f7disk

B.2 Future work

Based on the foundations laid by this proyect, some lines of future development are suggested.

First of all, incorporating new features into the kernel to complement the already implemented ones would be desirable. For instance, a serial port driver would enable the usage of a debugger from emulated environments. The bootloader must implement the multitrack loading to fully supporting VirtualBox, and integrate keyboard support on real mode to become into an interactive boot monitor. Also, it would be convenient to implement support for user processes and complete the kernel side of soft reset.

In the same vein, it is also expected the implementation of a file system. It is advised to base its implementation on the **9P2000** basic protocol of Plan 9 [22], and then consider minor modifications to improve it [23].

In second place, the directory tree accounts a potential reimplementation for new architectures. The Section 2.1 particularly shows how to install the toolchain for the 64-bit variant of the same architecture used by the project and, as both of them share many similarities, give it support in the same way is the next logical step.

In third place, the F7h partition can be improved to include the size of the used space of the slots, or even take advantage of the available space at the header's sector in order to use it as a boot sector. The standarization in that way would contribute to its reutilization by other compatible operating systems.

Lastly, the modification of the system and the creation of new implementations are encouraged [1]. It is suggested the use of languages as Go, Rust or others oriented to system software programming. Also the usage of other assemblers, whether exploring the AT&T branch with GNU Assembler or the Intel family with FASM.

Siglas

A

ACPI

Advanced Configuration and Power Interface (‘interfaz avanzada de configuración y energía’).

B

BIOS

Basic Input/Output System (‘sistema básico de entrada y salida’).

BSS

Block Started by Symbol (‘bloque comenzado por símbolo’).

C

CHS

Cylinder-head-sector (‘cilindro-cabezal-sector’).

E

EBDA

Extended BIOS Data Area (‘área extendida de datos del BIOS’).

ELF

Executable and Linkable Format (‘formato enlazable y ejecutable’).

G

GCC

GNU Compiler Collection (anteriormente «GNU C Compiler»).

GDT

Global Descriptor Table (‘tabla global de descriptores’).

GiB

Gibibyte (‘gigabyte binario’). Equivale a 1024 MiB.

H

HPET

High Precision Event Timers (‘temporizadores de eventos de alta precisión’).

I

IDT

Interrupt Descriptor Table (‘tabla de descriptores de interrupción’).

IRQ

Interrupt Request (‘petición de interrupción’).

ISR

Interrupt Service Routine (‘rutina de servicio de interrupción’).

IVT

Interrupt Vector Table (‘tabla de vectores de interrupción’).

K

KiB

Kibibyte (‘kilobyte binario’). Equivale a 1024 bytes.

L

LBA

Logical Block Addressing (‘direccionamiento de bloque lógico’).

LDT

Local Descriptor Table (‘tabla de descriptores locales’).

M

MBR

Master Boot Record (‘registro maestro de arranque’).

MiB

Mebibyte (‘megabyte binario’). Equivale a 1024 KiB.

MMU

Memory Management Unit (‘unidad de gestión de memoria’).

N

NASM

Netwide Assembler. Puede referirse tanto al ensamblador en sí como al lenguaje que implementa.

NMI

Non-Maskarable Interrupt (‘interrupción no enmascarable’).

P

PAE

Physical Address Extension (‘extensión de direccionamiento físico’).

PDE

Page Table Entry (‘entrada de la tabla de páginas’).

PDT

Page Directory Table (‘tabla del directorio de páginas’).

PDTE

Page Directory Table Entry (‘entrada de la tabla del directorio de páginas’).

PE (flag)

Protected Mode Enable.

PG (flag)

Paging Enable.

PIC

Position-Independent Code (‘código independiente de la posición’) o *Programmable Interrupt Controller* (‘controlador programable de interrupciones’).

PIT

Programmable Interval Timer (‘temporizador programable de intervalos’).

POST

Power-On Self-Test (‘autocomprobación de encendido’).

PT

Page Table (‘tabla de páginas’).

T

TSS

Task State Segment (‘segmento de estado de tareas’).

Glosario

C

compatible (PC)

Los compatibles con el **IBM PC** son similares a dicha computadora. Inicialmente denominaba a los ordenadores clónicos, que terminaron por reemplazarla. Su arquitectura ha evolucionado desde entonces.

H

huge pages

También conocidas como *super pages* o *large pages* en función del sistema operativo, se refiere a páginas de memoria cuyo tamaño supera el convencional de 4 KiB.

L

little-endian

En un número representado en formato little-endian, el orden de los bytes va del menos significativo al más significativo. El término suele ser motivo de confusión: debe entenderse que el '*extremo* pequeño' va primero, y no que sea 'de *final* pequeño'. 20

R

round-robin (planificación)

Algoritmo de planificación en el que los procesos se organizan en una cola de prioridad circular. El planificador la recorre asignando una fracción de tiempo de ejecución de la CPU..

W

wrapper (rutina)

Normalmente se refiere a rutinas que «envuelven» a una secuencia de instrucciones escritas en ensamblador con el fin de poder invocarlas desde lenguajes de alto nivel.

Bibliografía

[1] R. Pike, *System Software Research is Irrelevant*, http://www.herpolhode.com/rob/utah2000.pdf, diapositivas de la charla, feb. de 2000.

[2] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey y P. Winterbottom, *Plan 9 from Bell Labs*, 1995.

[3] F. J. Ballesteros, *Introduction to Operating Systems Abstractions*, http://lsub.org/who/nemo/9.intro.pdf, borrador, sep. de 2007.

[4] A. S. Tanenbaum y A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005, ISBN: 0131429388.

[5] *Repositorio de GCC (resumen)*, https://gcc.gnu.org/git/?p=gcc.git;a=summary, revisado por última vez en dic. de 2019.

[6] *GCC Cross-Compiler*, https://wiki.osdev.org/GCC_Cross-Compiler, revisado por última vez en dic. de 2019.

[7] *Variables Used by Implicit Rules*, https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html, revisado por última vez en dic. de 2019.

[8] *Quick Start for MASM Users (NASM)*, https://nasm.us/doc/nasmdoc2.html#section-2.2, revisado por última vez en enero de 2020.

[9] *Página de NASM*, https://www.nasm.us/, revisado por última vez en enero de 2020.

[10] *Chapter 6: Assembler Directives (NASM)*, https://www.nasm.us/doc/nasmdoc6.html, revisado por última vez en dic. de 2019.

[11] *NASM Command-Line Syntax*, https://www.nasm.us/doc/nasmdoc2.html#section-2.1, revisado por última vez en dic. de 2019.

[12] *BOCHS, the Cross Platform IA-32 Emulator*, http://bochs.sourceforge.net/.

[13] *QEMU/Monitor*, https://es.wikibooks.org/wiki/QEMU/Monitor, revisado por última vez en dic. de 2019.

[14] R. Pike y col., *Plan 9 Programmer's Manual: Volume 1*. 2002. Dirección: https://9p.io/sys/man/vol1.pdf.

[15] R. Pike, *Notes on Programming in C*, http://www.literateprogramming.com/pikestyle.pdf, revisado por última vez en dic. de 2019.

[16] *Sizes Should Only Span Unsigned*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1089r0.pdf, revisado por última vez en dic. de 2019.

[17] *GCC, Options for Linking (-nostdlib)*, https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/Link-Options.html#index-nostdlib, revisado por última vez en dic. de 2019.

[18] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer Manual - Volume 3A*, 253668-060US. 2016.

[19] Intel, *IA-PC HPET (High Precision Event Timers) Specification*, https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf, revisado por última vez en enero de 2020.

[20] *Advanced Configuration and Power Interface Specification, Revision 3.0b*, https://uefi.org/sites/default/files/resources/ACPI_3_Errata_B.pdf, 2006.

[21] W. Mauerer, *Professional Linux Kernel Architecture*. GBR: Wrox Press Ltd., 2008, ISBN: 0470343435.

[22] *RFC and standards documents relating the 9P protocol (and its extensions and applications)*, https://github.com/ericvh/9p-rfc, revisado por última vez en enero de 2020.

[23] F. J. Ballesteros, *A new mount table and protocol to make 9 and Nix faster*, http://lsub.org/export/9pix.pdf, revisado por última vez en enero de 2020.